Universitatea "POLITEHNICA" Bucureşti
Facultatea de Electronică, Telecomunicaţii şi Tehnologia Informaţiei

Teză de doctorat

Caracterizarea, gestionarea şi monitorizarea reţelelor de calculatoare ale sistemului de achiziţie de date ATLAS

Characterizing, managing and monitoring the networks for the ATLAS Data Acquisition System

Doctorand: **ing. Matei Dan CIOBOTARU**

Conducător ştiinţific: **Prof. dr. ing. Vasile BUZULOIU**

2007

Abstract

Particle physics studies the constituents of matter and the interactions between them. Many of the elementary particles do not exist under normal circumstances in nature. However, they can be created and detected during energetic collisions of other particles, as is done in particle accelerators.

The Large Hadron Collider (LHC) being built at CERN will be the world's largest circular particle accelerator, colliding protons at energies of 14 TeV. Only a very small fraction of the interactions will give raise to interesting phenomena. The collisions produced inside the accelerator are studied using particle detectors.

ATLAS is one of the detectors built around the LHC accelerator ring. During its operation, it will generate a data stream of 64 Terabytes/s. A Trigger and Data Acquisition System (TDAQ) is connected to ATLAS – its function is to acquire digitized data from the detector and apply trigger algorithms to identify the interesting events. Achieving this requires the power of over 2000 computers plus an interconnecting network capable of sustaining a throughput of over 150 Gbit/s with minimal loss and delay.

The implementation of this network required a detailed study of the available switching technologies to a high degree of precision in order to choose the appropriate components. We developed an FPGA-based platform (the GETB) for testing network devices. The GETB system proved to be flexible enough to be used as the basis of three different network-related projects. An analysis of the traffic pattern that is generated by the ATLAS data-taking applications was also possible thanks to the GETB.

Then, while the network was being assembled, parts of the ATLAS detector started commissioning – this task relied on a functional network. Thus it was imperative to be able to continuously identify existing and usable infrastructure and manage its operations. In addition, monitoring was required to detect any overload conditions with an indication where the excess demand was being generated. We developed tools to ease the maintenance of the network and to automatically produce inventory reports. We created a system that discovers the network topology and this permitted us to verify the installation and to track its progress. A real-time traffic visualization system has been built, allowing us to see at a glance which network segments are heavily utilized.

Later, as the network achieves production status, it will be necessary to extend the monitoring to identify individual applications' use of the available bandwidth. We studied a traffic monitoring technology that will allow us to have a better understanding on how the network is used. This technology, based on packet sampling, gives the possibility of having a complete view of the network: not only its total capacity utilization, but also how this capacity is divided among users and software applications.

This thesis describes the establishment of a set of tools designed to characterize, monitor and manage complex, large-scale, high-performance networks. We describe in detail how these tools were designed, calibrated, deployed and exploited. The work that led to the development of this thesis spans over more than four years and closely follows the development phases of the ATLAS network: its design, its installation and finally, its current and future operation.

Acknowledgments

I am grateful to all the people who contributed in one way or another to the completion of this thesis. First of all, I have to thank Prof. Vasile Buzuloiu who honored me with his precious professional guidance throughout the development of my research, in his unmistakable friendly manner. He also gave me the opportunity to come and work at CERN.

Then I would like to thank our group leader, the late and regretted Prof. Bob Dobinson, for his confidence and words of encouragement, especially during my first years at CERN.

A very special thanks goes to my direct supervisor, Brian Martin. His unique "management style" is something to be admired. Brian was a great resource of engineering knowledge and good advice in all matters of life. I have appreciated all his efforts aimed at keeping my material clear and understandable.

I wish to thank also to the entire ATLAS TDAQ community, most notably to David Francis, Krzysztof Korcyl and Andrew J. Lankford. Without the support from Prof. Lankford, this thesis would still be "work in progress".

Then I owe a great debt to all my colleagues from the ATLAS Networking Group. I would like to thank my friend and colleague Ştefan Stancu, who was always available for answering my questions. We had many fruitful discussions together and some of those have turned into results presented in this thesis.

A warm "thank you" goes to my colleague, Micheal LeVine, whose wisdom has always been highly appreciated. I learned a lot from Micheal – his analytical skills and sometimes his sharp critique have added value to this work.

I would also like to thank to Mihai Ivanovici and Răzvan Beuran for the lively and funny atmosphere they created at the office. Mihai takes credit as well for parts of the work described in this thesis and for proofreading the manuscript. And I cannot forget to say thanks to all the other extraordinary people I was lucky to meet during my stay at CERN: Gilad Kent, Laura Rădulescu, Lavinia Dârlea, Silvia Bătrâneanu, Marius Malciu, Alexandra Oltean, Cătălin Meiroşu and the Leahu brothers (Marius and Lucian). Many joyful moments, including debates upon the meaning of life, took place within this group.

And finally, I would like to express my gratitude to my wonderful parents who have shown a great deal of patience and have always been available for moral support along all these years.

Contents

Part I

Prologue

Chapter 1

Introduction

This thesis presents the contributions of the author to the development of the computer networks supporting the data acquisition system of the ATLAS experiment. This work has been done during the years spent by the author at CERN as part of the ATLAS Networking Group.

1.1 About CERN

CERN, the European Organization for Nuclear Research[1] [1], was founded in 1952 and is now the world's largest particle physics laboratory. Its main function is to provide the particle accelerators and other infrastructure needed for high energy physics research. Numerous experiments have been constructed at CERN by different international collaborations[2].

Nowadays, most of the activities at CERN are directed towards building a new accelerator, the Large Hadron Collider (LHC) and the experiments for it. Its operation is expected to start in the spring of 2008. The LHC will be the world's largest particle accelerator, developing the highest energy to date. The collider is contained in a 27 km circumference tunnel located at 100 m underground (Figure 1.1).

The collider tunnel contains two pipes enclosed within superconducting magnets cooled by liquid helium. Each pipe carries a proton beam. The two beams travel in opposite directions around the ring. Additional magnets are used to direct the beams to four intersection points where interactions between them will take place.

The colliding protons will have an energy of 7 TeV, giving a total collision energy of 14 TeV. In absolute terms, this is a small amount of energy: 1 TeV is about the energy of motion of a flying mosquito. The difference is that the LHC concentrates the energy into a space a million times smaller than a mosquito. This energy density is similar to what existed a few moments after the "big bang". The energy stored in the magnets that guide the proton beams is 11 GJ,

[1]The CERN acronym comes from "Conseil Européen pour la Recherche Nucléaire" – this was the original name, when the institute was founded in 1952.

[2]CERN has 20 member states. It collaborates with over 200 laboratories and universities from non-member states.

the equivalent of 2.5 tons of TNT. The power consumption of the LHC is 120 MW, almost equal to the power consumption of all the households in the entire Geneva area.

The protons inside the LHC do not form continuous beams. Instead, they will be "bunched" together into approximately 2800 bunches, so that interactions between the two beams will take place at discrete intervals never shorter than 25 ns. The maximum collision rate is $\frac{1}{25\text{ns}} = 40$ MHz. The interactions between protons give rise to energy and different types of particles.

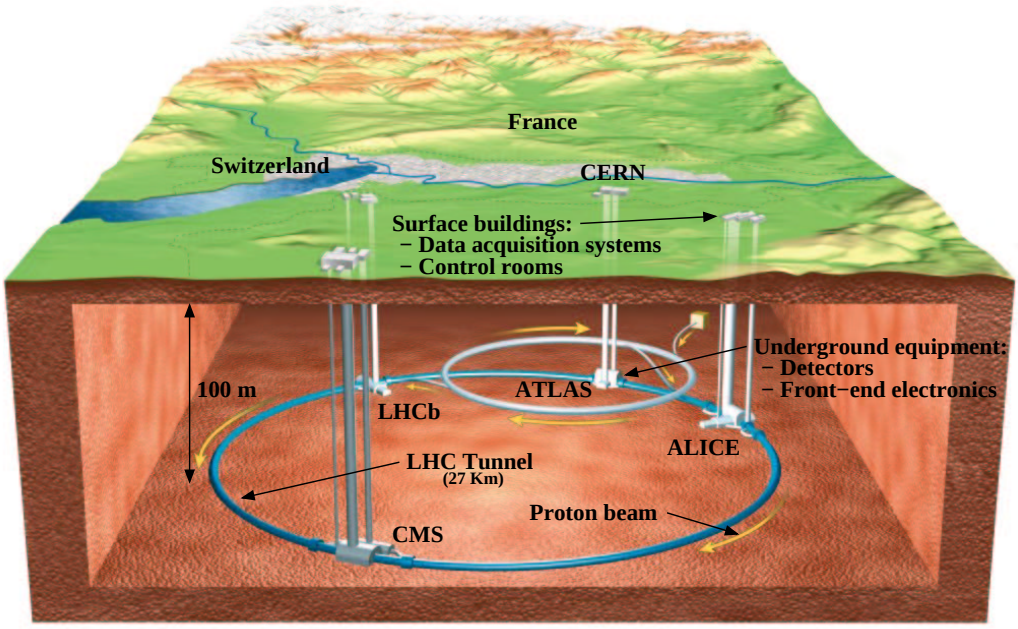


Figure 1.1: The Large Hadron Collider at CERN.

The most comprehensive model of particle interactions available today is known as the *Standard Model*. With the important exception of the Higgs boson, all of the particles predicted by the model have been observed. This model however, does not apply to energies greater than 1 TeV. As the LHC will work at 14 TeV, it is hoped that the experiments will provide enough data for physicists to develop a new theory that applies to higher energies and is able to predict the particles that can appear with all their properties: masses, momenta, energies, charges and nuclear spins.

Six detectors are being constructed at the LHC. They are located underground, in large caverns excavated at the LHC's intersection points. Two of them, ATLAS and CMS are "general purpose" particle detectors. The other four (LHCb, ALICE, TOTEM, and LHCf) are smaller and more specialized. Each of them will study particle collisions from a different point of view, and with different technologies. The detectors are installed at the four intersection points, as shown in Figure 1.1. ATLAS, CMS, ALICE and LHCb are located at beam crossing points. TOTEM shares the point with CMS, while LHCf is in the vicinity of ATLAS.

1.2 The ATLAS experiment

The ATLAS experiment (A Toroidal LHC ApparatuS), once completed, will have the largest detector[3] ever built at a particle collider (Figure 1.2). Rather than focusing on a particular physics process, ATLAS is designed to measure the broadest possible range of signals.

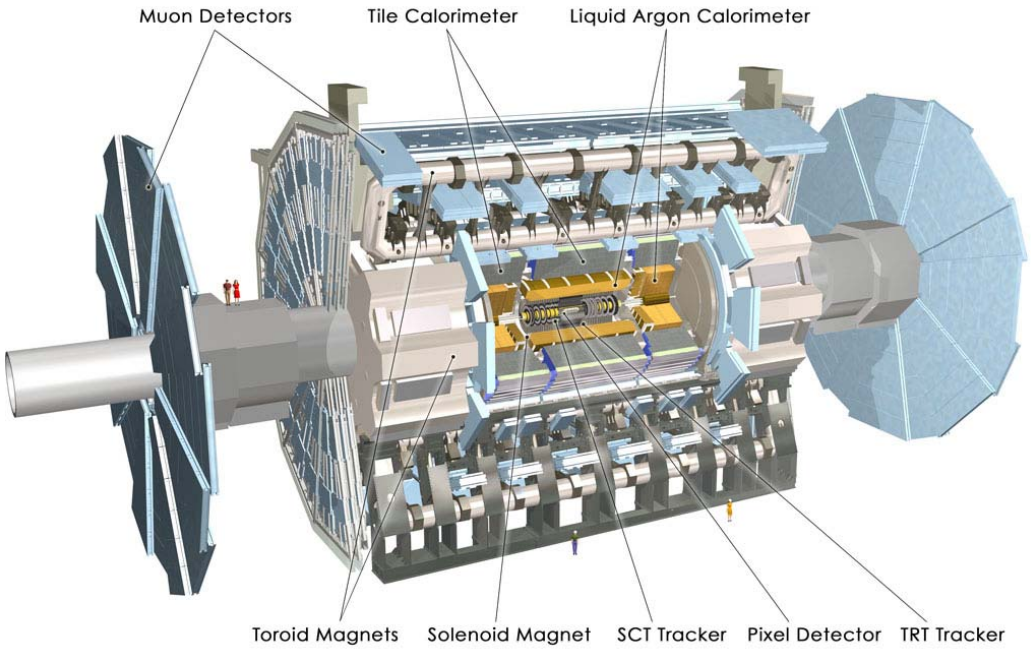


Figure 1.2: The ATLAS detector.

ATLAS consists of several concentric cylindrical layers built around the LHC beam intersection point. The layers are made up of sub-detectors of different types, each one being tailored to observe specific types or properties of particles. The different features that particles leave in each layer of the detector allow for effective particle identification and accurate measurements of energy and momentum.

Inside the collider, interactions are taking place at a rate of 40 MHz. Only a small fraction of the collisions produce interesting effects from the physics point of view. Each collision generates a cylindrical "image" at the level of the detector. This represents what we call an "event"; approximately 1.6 Mbyte of data are associated to each event[4]. A trigger and data acquisition system (TDAQ) is connected to the detector and its goal is to first identify and then save the interesting events, separating them from the rest of the "background noise".

The trigger system uses little information to identify, in real time, the most interesting events out of the 40 million beam crossings that occur every second in the center of the detector. There are three trigger levels: the first one based on detector electronics, while the other two run on a large computing cluster. After the first-level trigger, about 100000 events per second are selected.

[3]The dimensions of ATLAS are: 46 m long, 25 m wide and 25 m high. Its weight is of 7000 tons.
[4]The event contains the data recorded simultaneously by each of the sub-detectors.

After the third-level trigger, a few hundred events remain to be stored for further analysis. This amount of data will require over 300 Mbyte of disk space per second. In Chapter 2 we give a detailed overview of the TDAQ system.

1.3 Outline of the thesis

This thesis is focused on the implementation of the ATLAS TDAQ computer network; therefore the next chapter contains a presentation of the DAQ system and the network on which it is based. The rest of the work is divided into three parts which correspond to the development phases of the TDAQ network. For each part, the contributions of the author are described.

Part I – Design Due to the unusually demanding requirements, the complete design of the TDAQ network took quite a long time. It began with a survey of networking technologies (Section 2.3.2). Once the choice was made (in favor of Gigabit Ethernet), we had to draft an architecture (Section 2.4) and then try to map it on the products available on the market. This was a real challenge, mainly because most of the consumer-grade devices are not aimed for high performance environments like the TDAQ. Starting from the application's needs, we defined a list of required features for the devices and then started a market survey program. In order to test the ability to meet these requirements, we developed our own testing equipment. The system we used to evaluate the products is described in detail in Chapter 3. A part of the chapter is dedicated to the presentation of the underlying FPGA-based platform. We'll give details regarding the choice of components, the hardware architecture, the FPGA firmware and the control software. Another part of the initial design phase was a study of the traffic pattern generated by the ATLAS TDAQ system. In Chapter 4 we present an analytical model of this pattern, which is backed up by experimental measurements. This model can be used to predict the usage of the network as a function of the configuration of the TDAQ.

Part II – Installation Device acquisition and installation began after the completion of the design. Each device had to be configured and then checked for proper functionality. The installation had to be verified and compared to the initial plans. Each network connection had to be documented. Due to the size of the network, these tasks required automation. Chapter 5 brings into attention aspects related to network maintenance. Then the chapter presents the contributions of the author, namely methods and tools for the discovery of the physical network topology and for the configuration of the devices.

Part III – Operation Once the experiment is started, the network will enter the operation phase. From that moment on, it is supposed to function almost flawlessly, with minimal downtime, not to interfere with the data taking. Constant health monitoring is necessary: the traffic and all connections need to be watched and any anomalies spotted and promptly fixed. In Chapter 6 we discuss a traffic monitoring technology and its applications. This technology is based on statistical sampling and allows the identification of the applications which are consuming most of the network bandwidth (by studying the network traffic they generate).

In the following chapters we'll concentrate on the areas where the author has contributed most. For each part we shall try to add as much detail as required in order to give the reader a complete picture: when necessary, the work of other people will be described, the authors and the relevant papers being clearly cited. We have to stress the fact that most of the work behind this thesis has been done as part of a larger team. The last chapter (the 7^{th}) contains a summary of the author's achievements and a few directions for future research in network management.

Chapter 2

The ATLAS Data Acquisition System

The Trigger and Data Acquisition system (TDAQ) is the part responsible for the filtering and the preparation for archival of the events captured by the ATLAS detector. The architecture of the TDAQ is presented in this chapter, with a detailed description of the TDAQ network.

2.1 A three-layer approach

The ATLAS detector generates a huge amount of data – events are occurring at a rate of 40 MHz, each one having a size of 1.6 Mbyte; this yields a data rate of 64 Tbyte/s. As persistent real-time storage at this rate is not possible and because, on average, only a single event in a million is "interesting", a filtering mechanism has to be implemented, in order to save only the "most promising" events. The storage technology available at CERN can record at a rate of 320 Mbyte/s or 200 Hz[1]. So the DAQ system must be able to "down-sample" the data stream from the detector by a factor of 1:200000. This is easier to achieve by employing several cascaded filters[2]. With this observation in mind, we present in the text that follows, the organization of the TDAQ (Figure 2.1) and the interactions between its main components.

Level 1 Trigger This is the front-line of the TDAQ and interfaces directly to the detector. The Level 1 runs at the proton-proton bunch crossing rate of 40 MHz. Simple and fast algorithms are implemented in custom electronics – they have to reach a decision in 2.5 μs, at most[3]. For positive hits, the events are allowed to enter into the *Read Out Buffers (ROB)*. Simultaneously, information about the parts of the detector that were used to reach the decision are transmitted as *Regions of Interest (RoI)* to the second level of triggering. The Level 1 reduces the data rate from 40 MHz down to 100 kHz.

[1]Rates expressed in Hertz refer to the number of events transferred per second.

[2]Current computing technology cannot handle 64 Tbyte/s of data with the precision required for accurate event selection. This is why ATLAS has opted for a chain of filtering layers. Each layer makes a compromise between the accuracy of the operation and the time required to complete it.

[3]The Level 1 works synchronously with the detector. It has to take a decision every 25 ns ($\frac{1}{40\text{MHz}}$). The Level 1 has a pipeline of operations that it applies to an event. The total latency of this pipeline is 2.5 μs [2].

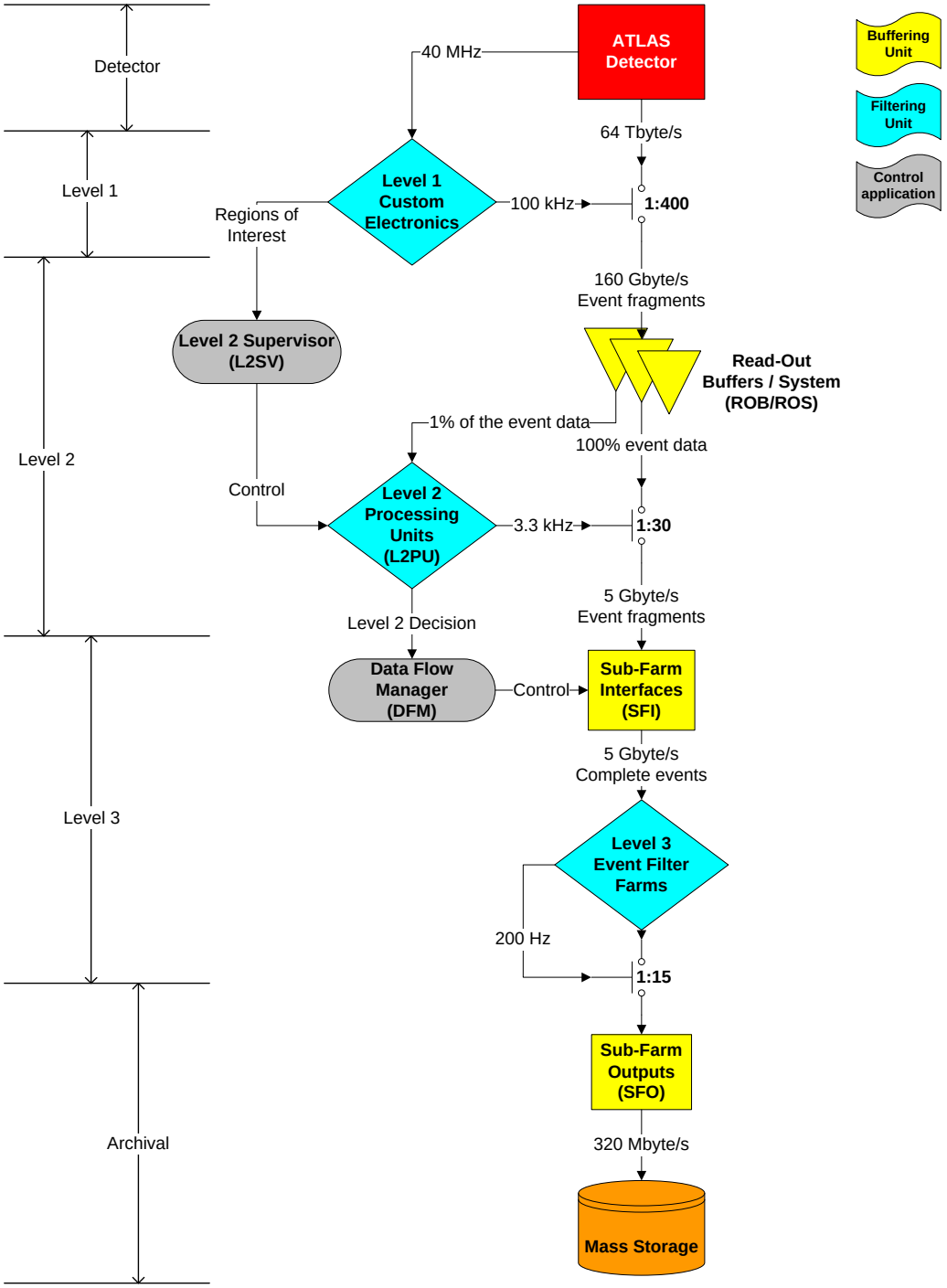


Figure 2.1: The TDAQ system with its three filtering layers.

Read Out System There are 1600 Read Out Buffers that hold *information fragments* about an event. The ROBs are implemented on PCI cards hosted in computers denoted as *Read Out Systems (ROSs)*. As the ROBs are physically connected to different sub-detectors, the data about a single event is scattered among all the buffers. There are approximatively 12 ROBs inside each ROS computer.

Level 2 Trigger This level is implemented in software running on a cluster of 500 computers: the *Level 2 Processing Units (L2PUs)*. The computing farm is controlled by several *Level 2 Supervisors (L2SVs)*. The L2PUs use the RoIs from Level 1 to retrieve only a part of the event data from the relevant ROBs. An L2PU has to decide whether to accept an event in about 10 ms. Once the decision taken, one of the supervisors (L2SV) is informed and the event is passed onward to another sub-system, the *Event Builder (EB)*. The rate of positive decisions from the Level 2 should not exceed 3.3 kHz.

Event Builder The L2SV forwards the Level 2 decision to the *DataFlow Manager (DFM)*, a software program which coordinates the assembly process of the validated events. The DFM assigns a *Sub-Farm Interface node (SFI)* to gather all the data available for an event selected by Level 2. The SFI asks the ROBs to deliver all the 1600 event fragments. After a full event is stored in the SFI, the DFM sends a "clear" message to all the ROBs, instructing them to free their buffers[4]. The SFI is the entry point to the third level of filtering, the *Event Filter*. As the EB sub-system does not filter the events, its output rate coincides to that of the Level 2, i.e. 3.3 kHz.

Level 3, the Event Filter (EF) This is the last level of filtering. A farm of about 1600 computers – the *Event Filter Processors (EFP)* – is used to analyze complete events using more complex and accurate physics algorithms[5]. The events are retrieved from the SFIs and processed; a fraction of them will be sent to the *Sub-Farm Outputs (SFOs)*. The SFOs are computers with large hard drives and are used to provide a smooth flow of constant bit rate (320 Mbyte/s) to the mass storage facilities. The SFOs complete the TDAQ chain.

We observe that the Level 1 trigger is the only part of the TDAQ that works synchronously with the detector. The rest of the TDAQ is implemented in software on commodity computers. As the performance of a computing cluster it's not fully predictable[6], there is an intermediate level of buffering in the Read Out System. The fragments of an event which passed the Level 1 tests are stored in the Read Out Buffers. A few remarks need to be made about the ROBs (and the corresponding Read Out Systems):

- For complete information about an event, fragments from all the 1600 ROBs are required.
- The ROBs do not delete fragments by themselves. The memory allocated for an event is cleared only upon reception of a "clear" message from the DFM.

[4]The DFM also sends "clear" messages for events rejected by the Level 2.

[5]An EFP has about one second to process an event. This is why it can afford to execute more complicated algorithms, in comparison to the Level 1 and Level 2.

[6]The performance of a computer depends on many factors: operating system, running processes, available memory, amount of activity on the system bus, etc. On the average, the performance level can be predicted; it is the instantaneous values that are unpredictable. This may lead to service unavailability for short time periods.

- If the ROBs run out of memory, they will eventually stall the DAQ process. It is possible that the ROBs fill-up because the Level 2 and Level 3 sub-systems are not fast enough. In this case a back-pressure mechanism is activated; this mechanism tries to reduce the Level 1 accept rate.

The main point is that the data in the ROBs has to be used and discarded fast enough, to make room for new events. This implies fast algorithms in the L2PUs and fast data transfers to the SFIs. The network used for these transfers has the most strict requirements in terms of performance.

The events accepted by Level 2 are stored in the SFI. The L2PUs are designed to work on incomplete event data, while the EFPs need full events. Another level of buffering is implemented in the SFIs. As opposed to the ROBs which were buffering event fragments, the SFIs store entire events. They are delivered on request to the EF processors (Level 3).

The operation of the entire TDAQ is controlled and monitored using the *Online Software*. This provides infrastructure services for information and error reporting, real-time monitoring and the ability to start and stop the TDAQ (or parts of it). Most of the *Online* services are accessible from a Graphical User Interface (GUI).

2.2 Traffic patterns

The nodes of the TDAQ computing farm communicate over the network using IP-based request-response protocols[7]. The nodes holding data (ROSs, SFIs) act as servers, while the nodes which process data behave as clients (L2PUs, SFIs, EFPs)[8]. The clients send request messages over the network to the servers; these respond with messages containing data. Chapter 4 contains an in-depth analysis of the request-response traffic. In the following we offer a high-level overview of the different traffic patterns that can be found between the layers of the TDAQ.

Level 1 filtering The Level 1 is directly connected to the ATLAS detector. Data flows into the Level 1 electronics and then a fraction of it enters the Read Out Buffers. These very high-speed transfers take place on dedicated links, they are not using the TDAQ network.

Level 2 traffic The Level 2 traffic is generated by the L2PUs requesting the data from a set of ROBs (the set being a function of the Regions of Interest). As only a fraction of the event is needed (1%) for the Level 2 analysis, the amount of data transfered is small. On average, the L2PU has to ask for 30 ROB fragments from approximatively 15 ROS computers. We call this an "N-to-1" pattern, with $N < 20$.

[7]Please note that in this work we assume the reader is familiar with computer networks and the Internet protocols. A quick introduction to the basics of Ethernet and TCP/IP can be consulted in Appendix A. The more advanced concepts will be explained when necessary in the main body of the text.

[8]The SFI is both a server and a client. It acts as a client when it assembles events and requests data from the ROSs. It is a server when it delivers the events to the Event Filter Processors.

Label	Pattern	Sources	Destinations	Data	Bandwidth
Raw data	1 to 1	Detector	Level 1 filters	Event fragments	512 Tbit/s
Level 1 trigger	1 to 1	Detector	Read Out System	Event fragments	1.28 Tbit/s
Level 2 filter	N to 1, $N < 20$	ROS (1%)	Level 2 Proc. Units	Event fragments	40 Gbit/s
Event building	N to 1, $N \approx 150$	ROS (100%)	Sub-Farm Interfaces	Event fragments	44 Gbit/s
Event filter	1 to 1	SFI	Event Filter Proc.	Full events	44 Gbit/s
Mass storage	1 to 1	EFPs	Sub-Farm Outputs	Full events	3 Gbit/s

Table 2.1: Traffic patterns in the ATLAS TDAQ.

Event Building traffic In order to build an event, the SFI has to get data from all the ROBs. At the network level, we have a large number of sources (1600 ROBs or 150 ROSs) sending data at the same time to a single destination (one SFI). We call this a "funnel" pattern. It is still "N-to-1", but now N is much higher ($N = 150$) in comparison to the Level 2 traffic. Because of this, in order to avoid data loss in the network, the traffic rate needs to be kept under control. A traffic shaping mechanism will be studied in Chapter 4.

Event Filter traffic An Event Filter Processor has all the information it needs in a single SFI. So the network has to sustain many "1-to-1" conversations in the case of Level 3 traffic.

Transfer to mass storage The output of the event filter goes to mass storage via a small number of SFO machines[9]. This involves again "1-to-1" transfers, but at a much lower rate.

Table 2.1 summarizes the above observations and includes the bandwidth requirements for each class of traffic (source: [3]). The section that follows contains the main requirements that were imposed to the TDAQ network.

2.3 The TDAQ network

The proper operation of the ATLAS TDAQ requires a fast and reliable network for transferring the messages. A failure in the network can affect the physics research (data taking). In this section we describe how the network has evolved over time, the constraints that have driven its design and the technological choices.

2.3.1 Constraints and requirements

Any networked application desires "perfect" network conditions: zero loss and zero latency. For a real-time application like the TDAQ, these conditions become requirements:

Small latency The request-response protocols can be quite sensitive to the end-to-end network latency (as we shall see in Chapter 4). The time spent for handling an event can be divided

[9]The plan is to have only 10 SFOs with 30 Tbyte total storage capacity.

into two parts: one spent while transferring the event fragments between the various layers and one spent actually analyzing the data. It is highly desirable to minimize the first part, i.e. the transfer time.

Zero packet loss When fragments of an event are lost in the network, the filtering applications have to request them again[10]. Any loss induces retransmissions, which may lead to a severe drop in the efficiency of the data acquisition.

In [4], [3] and [5] the upper bounds in terms of loss and latencies are specified[11]. In addition to the technical requirements, the DAQ network technology has to take into account other aspects:

Long term support The life-time of the experiment is over 10 years. Long-term support from industry is essential in order to ensure future upgrades and smooth transitions to new technologies, when they become available.

Cost effectiveness CERN is a non-profit organization and has a limited budget available. The costs of the initial acquisition and then of the future maintenance of the network should be kept to a minimum.

The final rate at which the DAQ system will operate is going to be determined by many factors as explained next:

Available CPU resources The TDAQ works as a large distributed system with many computing nodes. As most of the event processing is done by computer software, the effective rate may be limited by the lack of CPU computing cycles.

Speed-optimized applications The developers of the TDAQ have put a lot of effort into optimizing the applications and tuning the algorithms for top performance. In addition, the compilers used have all the code optimization features enabled. With all these measures, there can still be flaws (memory leaks, unexpected error conditions) that can affect the operation of the DAQ and reduce the overall rate.

Application configuration Each individual program in the TDAQ has its own set of configuration parameters. As the applications cooperate, there are many inter-dependencies between the parameters. In the past, the configuration of the whole DAQ system required the combined knowledge of many experts. Without careful tuning, the DAQ would not even start. Recently, new tools have been developed to automatize the configuration process. Unfortunately, the possibility of mis-configuring the system still exists. During the commissioning phase of the DAQ, it was not uncommon to observe a low acquisition rate just because of incorrect configuration parameters.

[10] If it cannot obtain all the data describing an event, the L2PU will accept the event as it is. Data gathering applications (the SFIs) will build an incomplete event.

[11] For a switching device which is fully loaded, the latency between two ports should be less than 3 ms and the packet loss rate should be smaller than 0.0001%.

Available network bandwidth Finally, one of the most important factors that determines the DAQ rate is the amount of bandwidth available to transfer and analyze the events. The lack of network bandwidth will have a direct (negative) impact on the data acquisition.

Taking into account all these observations, it was necessary to find the networking technology that would best suit the TDAQ needs.

2.3.2 Choice of technology

A lot of studies have been carried out in order to find the best technology for the implementation of the TDAQ network. Three of the candidate technologies have been[12]:

ATM The Asynchronous Transfer Mode is a networking technology which supports speeds of up to 622 Mbit/s. It is mainly used in Wide Area Networks by telecommunication companies. Before the advent of Gigabit Ethernet, ATM was seriously considered to be used in the Level 2 and Event Building networks [6], [7].

Myrinet This is a network technology designed especially for high performance cluster computing. It features very low latency and, optionally, guaranteed delivery of messages. On the other hand, Myrinet is optimized for computing clusters, where nodes need to communicate constantly between them. In the ATLAS DAQ, the main flow is "vertical", from the detector down through the three filtering layers. There is little communication on the "horizontal" direction, between computers/applications having the same function. Myrinet was evaluated for the DAQ of the CMS experiment at CERN [8], [9].

Ethernet Around the year 2000, Ethernet, invented in 1974, started to become increasingly popular. This was due to an increase in performance (the standard for 1 Gbit/s approved in 1999), the introduction of the full duplex operation (1997), the star topology (using switches) and, maybe the most important factor, a continuous decrease in the prices of the inter-connect hardware. Ethernet was gaining wide acceptance from industry and this generally implies support and availability over the long term. Compared to ATM, Ethernet had a better ratio of performance to price.

The ATLAS community realized the advantages of Ethernet. After a series of studies meant to learn its possibilities and limitations ([10], [11], [12]), a first proposal of the TDAQ architecture has been made in [13] (2002). That proposal assumed parts of the network were running at 100 Mbit/s (Fast Ethernet).

In 2001 it became clearer that the TDAQ would most likely be based on Ethernet, as this was becoming the de-facto standard in local high-speed networks. Then we concentrated our efforts to see what kind of devices were available on the market and if they can handle the massive data transfers needed for ATLAS.

[12]We shall refer to studies done at CERN as a whole, not only for ATLAS. All the LHC experiments implement part of their DAQ systems on computer clusters.

In 2002 a paper was published [14] with results from the first tests on Ethernet equipment. These tests were done using tools developed at CERN. The network architecture was revised in 2003 [15], [16]. During 2004 and 2005, it became clear that Fast Ethernet would become obsolete by the time ATLAS starts. Therefore, we concentrated our research on the Gigabit Ethernet technology.

A complete system for estimating the performance of devices and determining their features has been designed and implemented (one of the main contributions of the author) (see [17] and Chapter 3). In parallel, we identified the required features for the ATLAS network devices, and complemented them by a set of testing methodologies [4].

After two years of studies[13], we published in 2005 an upgrade of the network architecture [18] – this was based entirely on Gigabit Ethernet (GE) and was considering the inclusion of 10 Gigabit Ethernet (10GE). The network design was finalized in 2006 [19]. Based on our recommendations with respect to the architecture and the devices, the network equipment has been acquired and installation has been started.

2.4 The architecture of the TDAQ network

The scale of the TDAQ system and the requirements in terms of rate (bandwidth) have driven the design of the network. The outcome of this process was a complete architecture, which is outlined below[14]. A detailed description can be found in [3].

2.4.1 Equipment

The ATLAS detector is located in an underground cavern, at 100 m below the surface. The components of the Level 1 filter and the Read Out Systems are installed in the close vicinity of the detector. The rest of the TDAQ, i.e. the Level 2 and Level 3 farms, are placed in a surface building, as shown in Figure 2.2.

The TDAQ network is entirely based on Ethernet technology. The network is built as a tree of switches and routers (Figure 2.3). The core nodes of the tree are chassis-based devices that support many connections and high throughput. The leaves of the tree consist of smaller devices (concentrators) which are used to connect the computers to the network. The two types of devices are described next.

Concentrator switches These devices have 24 or 48 GE ports and one or two 10GE uplink interfaces. While normally designed for Layer 2 switching, the latest models have Layer 3 routing capabilities. They are used to "concentrate" the links from a set of machines and provide connectivity via the uplinks to other parts of the network.

[13] We conducted a market survey which included measurements on more than 20 device models produced by over 10 vendors.

[14] This section contains material presented in [19].

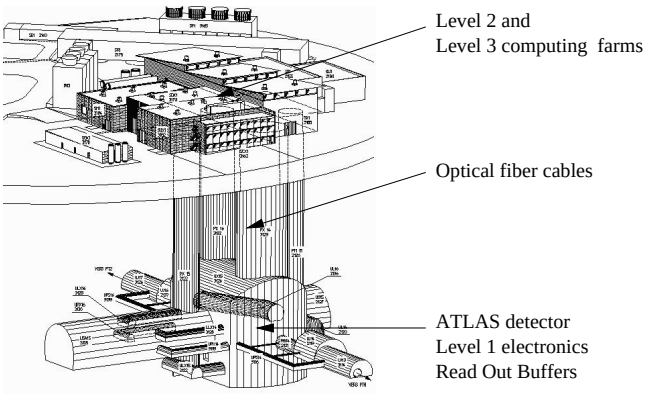


Figure 2.2: The ATLAS pit.

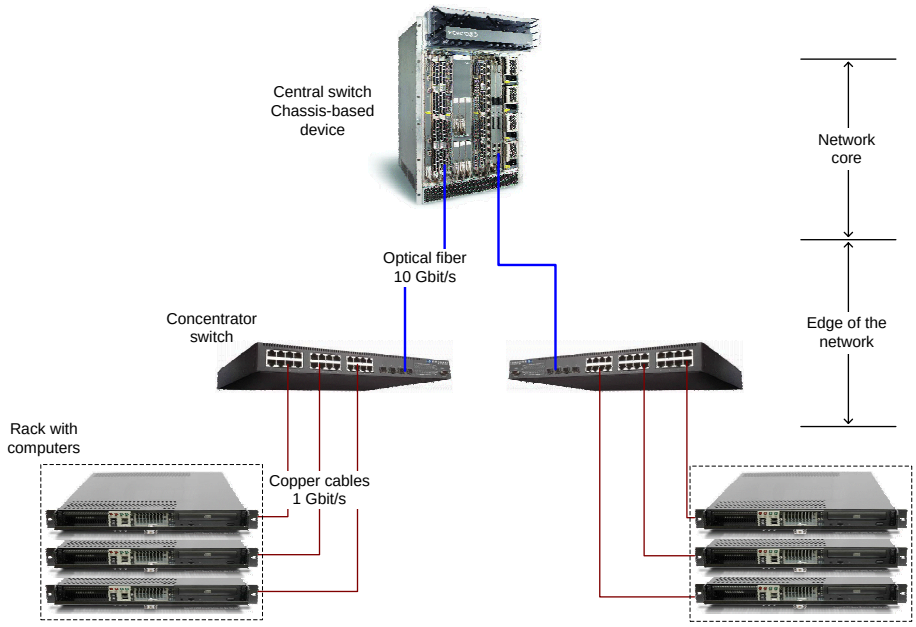


Figure 2.3: Equipment organization in the TDAQ network.

Chassis-based devices These are bigger devices with a large set of features. They are constructed around a chassis that supports a range of *line-cards* or *pluggable line modules*. A line-card can have up to 48 GE ports. Densities of up to 600 GE ports are possible within chassis-based devices. As they are targeted to mission-critical networks, they usually have built-in redundant components: power supplies, cooling fans, switching fabrics, etc. For chassis-based devices, it is common to have incoming links from the edges of the network, i.e. from the concentrator switches. However, computers with high demands of network bandwidth can be connected directly to these devices.

Towards the edges of the network, Gigabit Ethernet is used, while the links to the core are mostly based on 10 Gigabit Ethernet. The computers[15] have copper GE network interface cards (NICs). UTP Cat 6 cables[16] will be used to connect these to the network. Optical fibers are employed to connect the concentrator (edge) switches to the central devices. These links carry 10 Gbit/s traffic which is supported only on optical fiber media (850 nm, multi-mode fiber).

2.4.2 Layout

The architecture of the TDAQ system, including that of the network, is shown in Figure 2.4 (source: [19]). There are three sub-networks: two of them are for data transfers (i.e. physics events), while the third is for general-purpose communication (control and monitoring).

FrontEnd network This part of the network has the highest bandwidth requirements (Figure 2.5(a)). It carries data from the Read Out Systems to the L2PUs and SFIs, i.e. the Level 2 and the Event Building traffic streams. To provide redundancy, the core of this network will consist of two chassis-based devices[17]. When the two central switches are fully functional, they share the load; when one of them fails, the traffic is automatically routed to the other device[18].

BackEnd network The SFIs, the EFPs and the SFOs are connected to the BackEnd network. At its core it has a chassis-based switch with built-in redundancy. The EFPs are connected via concentrator switches to this core (Figure 2.5(b)).

Control network This network provides connectivity between any two computers inside the ATLAS TDAQ domain (Figure 2.6). With the exception of the event data transfers, all the communication between the TDAQ applications takes place over the control network. Services like shared file systems, database access, monitoring – they all run across this path. While the control network does not have very tight performance requirements, it has to be very reliable. For management purposes, the switches of the data networks are also linked to the control infrastructure.

[15] The ROSs, the SFIs and the Level 2 and Level 3 computing farms.
[16] Unshielded Twisted Pair, Category 6.
[17] Only one of them is installed at the time of this writing.
[18] The load balancing and the redundancy are implemented using standard protocols like Multiple Spanning Tree (MST) and Virtual Local Area Networks (VLAN). Details available in [19] and [20].

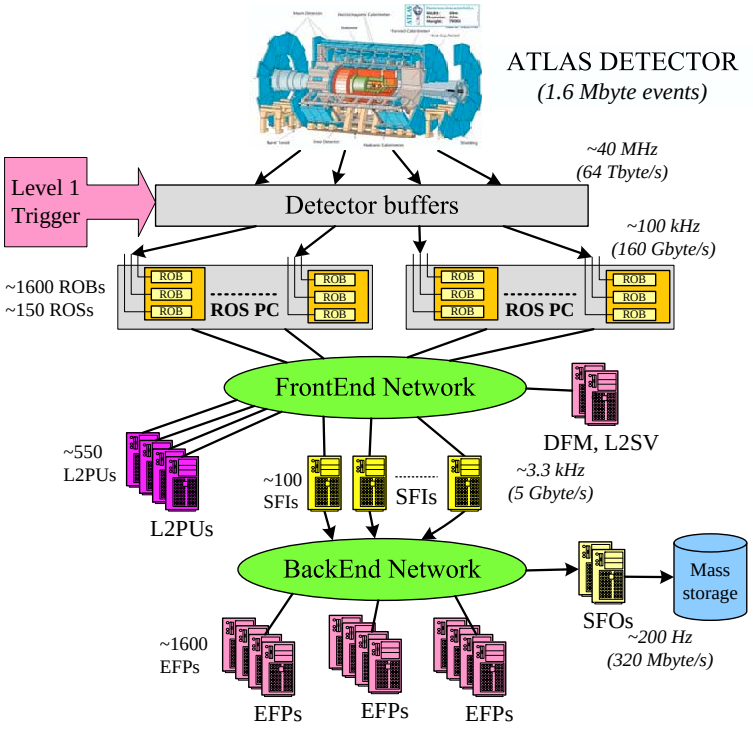
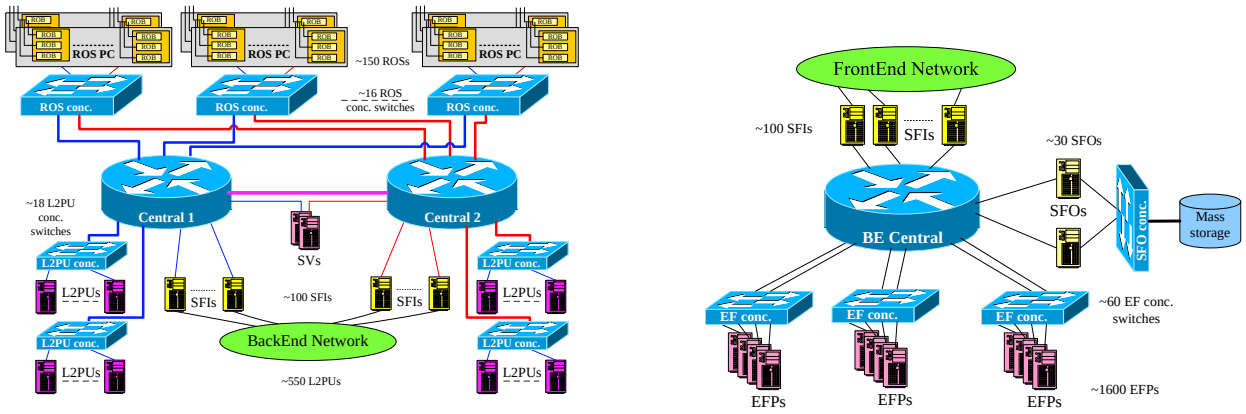


Figure 2.4: The TDAQ system and the underlying data networks – Overview.



(a) FrontEnd Network

(b) BackEnd Network

Figure 2.5: The TDAQ data networks – Detail.

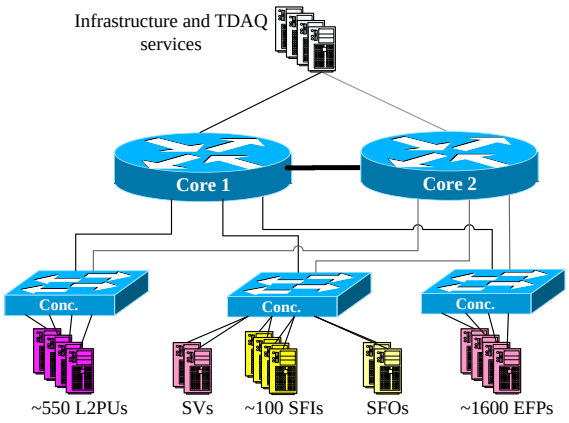


Figure 2.6: The TDAQ control network.

The computers are installed in racks[19]. Inside each rack there are one or two concentrator switches – one for the control and (optionally) one for the data network. These switches are used to reduce the costs of cabling and the size of the central switches. For example the Read Out Systems and the L2PUs are connected via 10G uplinks to the core of the network. The SFIs, because of their "N-to-1" traffic pattern (Section 2.2), are plugged directly into the central switches.

The network is designed to provide redundancy in case of failure of one of the core (central) devices. For the concentrator switches, a failure means that an entire rack becomes inaccessible. If a rack contains machines with the same function, a power or switch failure would give raise to an unbalanced DAQ system. Ideally, a rack should contain an uniform mixture of L2PUs, SFIs and EFPs. This would minimize the negative impacts in case of failures.

We explain next how each part of the TDAQ is connected to the network.

Read Out System The computers containing the Read Out Buffer cards are installed in racks in an underground room. Each rack has a concentrator switch for the FrontEnd data network and one for the control network. The data switches have 10G uplinks up to the core devices installed at the surface. The concentration ratio for the ROSs is 1:1 so there is no over-subscription[20]. There are 10 ROSs in a rack, each one having two GE network interfaces; the maximum ROS output (20 Gbit/s) matches the capacity of the uplinks (two times 10G). We can say we have 1:1 "concentration" in terms of bandwidth and 10:1 concentration in terms of connections (cables).

Level 2 Processing Units The L2PUs are installed at the surface. They need less network bandwidth, so the concentration ratio is 3:1. 30 L2PUs connect to a concentrator switch

[19]There are about 30 computers in a rack. With the exception of the ROSs which have a height of 4U, the rest of the computers are 1U industrial PCs. 1U = 4.445 cm.

[20]We say that a switch port is oversubscribed if the amount of egress traffic that has to go out of it, exceeds the capacity of the link. This can easily happen when we have multiple sources sending data to a single destination; a scenario which is very common in the TDAQ.

and then via one 10G uplink to the FrontEnd data core. The average occupancy on the L2PU 10G uplinks is expected to be less than 25%[21].

Sub-Farm Interfaces The SFIs are the most demanding applications, as they need to collect data from all the ROSs (and they perform *no* physics analysis). They are connected directly to the core devices, without any intermediate concentrator switches[22]. An SFI has links to both the FrontEnd and BackEnd core devices. On one link it receives event fragments from the ROSs, assembles them, and then on the other link it delivers full events to the EFPs.

Event Filter Processors The EFPs are connected via concentrator switches to the BackEnd core. The concentration ratio is 15:1 – 30 EFPs and two Gigabit uplinks. The ratio is so big because there are 1600 EFPs, i.e. 54 racks[23]. They have to absorb the 44 Gbit/s from the SFIs (see Table 2.1), so each rack should not need more than 1 Gbit/s. Two links of 1 Gbit were allocated as a safety factor and for redundancy[24].

Sub-Farm Outputs The SFO machines are connected on one side directly to the BackEnd core switch – this connection is used to get the events from the EFPs. Their second connection is to a switch that has a 10G uplink to the CERN mass-storage service.

Control applications The machines acting as DataFlow Managers (DFM) or Level 2 Supervisors (L2SV) are connected redundantly to the core switches.

The installation of the network has begun in 2006 and parts of it have already been commissioned. We shall see in Chapter 5 a "snapshot" of the current state of the TDAQ networks (page 138).

2.5 Summary

The ATLAS TDAQ system employs a large, distributed computing farm for filtering the collision events recorded by the detector. After providing a high-level description of the way the TDAQ works, we described the architecture of the underlying network. This network has to function at a constant speed of at least 150 Gbit/s; in addition, it has to support an asymmetric traffic pattern that can easily lead to hot spots (network congestion). The development of such a complex infrastructure spans over more than six years. We continue the presentation with one of the major challenges we had to overcome during the design phase: the selection of the best network equipment for the ATLAS TDAQ network.

[21] There will be 500 L2PUs in the final system. The total bandwidth expected to be used by the Level 2 trigger is 40 Gbit/s (Table 2.1). For one L2PU we have $\frac{40\text{Gbit/s}}{500\text{L2PUs}} = 80$ Mbit/s. The entire rack will need to receive from the ROSs: $30 \times 80 = 2.4$ Gbit/s.

[22] The reason for this is that the chassis-based devices have larger buffering capabilities and they can handle congestion better than concentrator devices.

[23] The number of EF processors was calculated as follows. The Level 2 layer accepts events at 3.3 kHz (3300 events/second). The time spent in the Level 3 (EF) algorithms is one second. Therefore we would need 3300 computers to analyze all the events. ATLAS acquired dual-core machines so approximatively 1600 nodes should be able to handle all the data.

[24] The two links operate as an Ethernet "trunk" [21].

Part II

Design of the TDAQ network

Chapter 3

Testing Ethernet devices

In Section 2.4 we've outlined the TDAQ network architecture and shown that it is based on switches and routers. With so many manufacturers and products on the market, we quickly understood we have to be very selective as not all of them were up to our demands. Therefore we designed and implemented a switch test equipment along with the appropriate testing methodology.

3.1 Network testing

A generic *tester* is a tool that can assess the performance and verify the functionality of a given *System Under Test* (SUT). The *system* in our case can be an entire network or a single network device (then we say we deal with a DUT – *Device Under Test*). The tester works by injecting artificial traffic into the SUT and monitoring the output (Figure 3.1). Ideally, a tester should emulate, as closely as possible, real applications traffic patterns, i.e. the artificial traffic should resemble as much as possible to what we have in a real network. Two techniques can be used: one is to emulate real applications that communicate over the network, the other is to generate the traffic according to some predefined patterns.

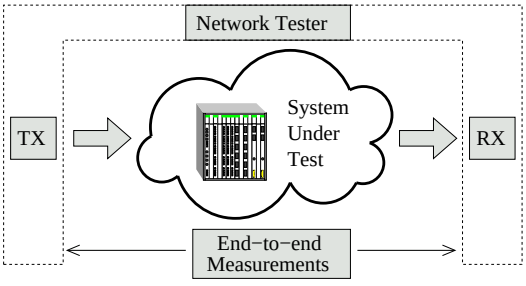


Figure 3.1: A network testing system.

A tester has to constantly watch over the behavior of the SUT, i.e. its output. Losses in the

network should be detected and quantified, delays should be measured and the input/output[1] data rates should be measured[2].

When doing an evaluation of many devices, one has to define a testing methodology and then run the same tests on all devices, comparing the results at the end. In this case, the feature of *scripting*[3] is mandatory, in order to automate the testing process. A reporting system is also desirable. Based on the test results it should produce plots and summary tables. One can envisage an expert system coupled to the report generation – this could try to find patterns in the results and draw conclusions about the device behavior.

3.1.1 Requirements

A set of performance metrics and test procedures has been defined in order to check the compliance of candidate devices for the ATLAS TDAQ network [4]. These procedures try to characterize the performance and functional aspects that will be relevant for the final ATLAS data-taking phase. Commercial network testing equipment such as the Ixia Optixia [24] or Spirent Smartbits [25] is mainly oriented towards protocol compliance and raw performance, lacking the flexibility needed in defining ATLAS-like traffic patterns. The following contains a summary of the main requirements we had with respect to the network tester:

Gigabit Ethernet The testing system had to fully support the Gigabit Ethernet standard. It was assumed of course that the tester would be able to send and receive packets at the full GE line-speed.

High port density The central devices in the TDAQ network have more than 100 GE ports. The tester had to scale to such high densities, without any loss in performance or functionality.

Flexible traffic patterns As explained in Section 2.2, we need to be able to emulate as closely as possible the traffic patterns that appear in the TDAQ. This is needed in order to create the environment that would have to be supported by the device when deployed in the final network. For example, the request-response pattern, which is specific to ATLAS, is a feature difficult to implement with commercial testers.

Fully programmable The testing platform that we envisaged had to be capable of changing its functionality, even completely if necessary. We wanted a system which could be adapted in the future, in order to fit entirely new requirements (e.g. applications like network emulation or monitoring).

[1] The *input* traffic is known as it is generated by the tester. The *output* traffic is the result of the interaction between the artificial traffic at the input, the inner mechanisms of the SUT and any other background traffic that passes through the SUT at the moment of the test.

[2] Instead of *rate* we shall use sometimes the term *throughput* or *speed*. They all refer to the quantity of data transferred in a given time interval.

[3] A *script* is a computer program written in an interpreted language. Scripts are generally used to perform simple tasks, but is not uncommon to find full applications written in scripting languages. Python [22] or Perl [23] are examples of scripting languages.

Automatic testing The tester was needed to complete a market survey involving many devices. Automatic runs with minimal user intervention are essential in this case. Graphical User Interfaces (GUIs) are useful, but not suitable for large scale testing.

These requirements indicated that a programmable platform would better suit our needs. Based on our past experience with hardware-accelerated networking applications ([11], [14]) we designed a new FPGA-based[4] platform called the *Gigabit Ethernet Testbed (GETB)* [17]. This platform is fully programmable and delivers Gigabit wire-speed performance. It can be used to create applications that generate or process Ethernet data in real-time.

In Section 3.2 we describe in detail the architecture of the GETB platform: its design process and then its physical and software implementations. Then we shall present the projects which are currently using it. Firstly, the Network Tester, the tool that has been used to evaluate switches for ATLAS, is presented in Section 3.3. Then in Section 3.4 we present sample results obtained using the tester. Finally, in Section 3.5 we briefly discuss two other projects which are based on the same hardware.

3.2 The GETB platform

The GETB uses custom-built PCI[5] cards and control software to provide a platform for Gigabit Ethernet applications. The hardware and software designs[6] are presented in this section.

3.2.1 Design process

The GETB project started with a list of requirements; the next step was to determine the implications these imposed on the hardware and software designs. Compromises were necessary in order to make a balance between costs, performance and ease of development.

As the GETB was a custom design and we were envisioning the production in small quantities, we wanted to have the hardware as simple as possible in order to reduce the chances of manufacturing errors. We needed a simple board layout and a small number of discrete electronic components.

The time-frame allocated for the GETB hardware design, the manufacturing and the software development was about one year. This constraint was imposed by the ATLAS switch evaluation program which was in preparation while the GETB system was being developed.

[4]Field Programmable Gate Array.

[5]Peripheral Component Interconnect.

[6]The design of the GETB hardware – component selection, schematics and board layout – were done at CERN by our colleagues, Mr. Jaroslav Pech and Mr. Brian Martin. The FPGA firmware was designed and developed by the author with contributions from Mr. Micheal LeVine. The GETB control software has been created by the author.

3.2.1.1 Form factor

The physical implementation of the GETB had to consider the main application, i.e. testing Ethernet equipment. This meant the GETB had to support a large number of Ethernet ports (more than 100, see the initial project proposal at [26]). If this were the only constraint, then the best option would have been a design similar to an Ethernet switch: a stand-alone device with more than 30 ports and an independent power supply. However, this approach is not suited for other types of applications which require just one or two Ethernet ports – examples are network emulation or traffic recording. In addition, a platform composed of a few high port density boxes has a low fault tolerance – for example, a power supply failure brings off-line more than 30 ports.

For the GETB we selected a different approach, with a much lower granularity. It was decided to implement the GETB as a PCI card. On each card we envisaged to have *two Ethernet ports*. It was foreseen to mount these cards into industrial PCs which can have up to twelve PCI slots. PCI was chosen because it simplifies the management of the cards (it is done from the host computer) and because it supports bandwidths of over 1 Gbit/s. The use of the PCI interface eases also the deployment (virtually any modern computer has several PCI slots).

Even though the PCI form factor allowed us to have up to four ports per card, we limited the number of ports to only two in order to reduce the complexity of the board and to decrease the load on the controller.

3.2.1.2 Controller

The most important design decision was related to the controller of the GETB platform. We had four options:

- A general purpose microprocessor (CPU)
- A generic Network Processor (NP)
- An Application Specific Integrated Circuit (ASIC)
- A Field Programmable Gate Array (FPGA)

From the point of view of the software development, a general-purpose microprocessor would have been the best choice. A CPU, however, requires external components for accessing the Ethernet and PCI interfaces – this would have increased the complexity of the board. Another issue with CPUs is related to their performance and accuracy of the measurements – as multi-tasking is often implemented using a time sharing mechanism under the supervision of the operating system, it is difficult to program a CPU to handle several *real-time* activities – in our case, the GETB had to manage simultaneous transmission and reception of packets at Gigabit line-speed (on two Ethernet ports).

Another alternative that was considered was a Network Processor (NP) [27]. These are integrated circuits optimized for packet switching applications – they are used to build network

routers and switches. An NP contains instructions and pipelines that can handle multiple packets in parallel. They cannot be easily programmed for tasks like packet generation, online histograms or packet capture.

The highest possible performance can be obtained with an ASIC. The main disadvantage of an ASIC relies in its very high costs of engineering (development) and then fabrication. These costs can only be justified for commercial products which are built for mass production. As we were designing for research and we were not planning to build more than one hundred GETB boards, an ASIC-based solution could not be economically justified.

3.2.1.2.1 Field Programmable Gate Arrays An FPGA is a good compromise between an ASIC and a CPU. This type of integrated circuit can be "programmed" in the sense that it can implement any combinatorial or sequential circuit[7]. Once programmed, the performance of an FPGA is close to that of an ASIC. The main advantage over a CPU is that inside an FPGA we can have many small digital circuits which work in parallel. It is not unusual to have more than ten parallel tasks "running" in an FPGA. The equivalent CPU-based system would need more than ten processors and all the communication infrastructure between them.

An FPGA consists of a matrix of *logic elements* (LEs). The most common type of LE consists of a 4-input Look-Up Table (LUT) connected to a flip-flop. This element can implement any logic function with 4 inputs; the output can be (optionally) registered. In an FPGA, the LEs are placed inside an array of programmable inter-connections. By programming the LEs and by enabling/disabling the connections between them, one can define an almost unlimited number of digital circuits. Modern FPGAs contain also storage elements (SRAM[8]) and dedicated Digital Signal Processing (DSP) blocks[9]. An introduction to FPGAs is available in [28].

The circuit embedded in the FPGA is defined using a *Hardware Description Language* (HDL). A HDL is used to describe in a textual format the behavior of a digital circuit. This description is then passed through a compiler which transforms it into a digital circuit ready to be implemented in the FPGA. The compilation consists of three steps: synthesis, technology mapping and the final place-and-route. The first step (*synthesis*) transforms the behavioral description into a digital circuit based on elementary logic gates. Then the circuit is translated via *technology mapping* into an equivalent circuit which is based only on logic elements (the ones found in the FPGA). The final step, the *place-and-route* operation, is the most complex. It selects the *position* of the logic elements in the FPGA and then finds the optimal *routes* (inter-connections) between them.

[7] A *combinatorial* circuit is the physical implementation of a boolean logic function in which the outputs are uniquely determined by the current value of the inputs. A circuit whose outputs depend also on the previous input values is called *sequential*. Sequential circuits are built using storage elements (flip-flops, memories). Most practical digital circuits contain a mixture of combinatorial and sequential logic. For example, in order to implement a state machine, a circuit needs to calculate the next state using combinatorial logic and to store it in a register. Virtually all digital circuits in use today are *synchronous*, meaning that any change in the state of the internal registers is triggered by the edge of a clock signal. Synchronous circuits are so popular because they are easier to design. The only requirement for them to function reliably is that the combinatorial logic determining the contents of registers must provide a stable result during one clock period.

[8] Static Random Access Memory.

[9] A DSP block contains circuits for multiplication and division.

The three steps described above are performed in a sequence by a program called a *fitter* (because it tries to "fit" a logic circuit in the FPGA). The circuit that is programmed in the FPGA is called the *firmware*.

The fitter tries to pack as much logic into as few logic elements as possible (area optimization). It also tries to minimize the delays between the various logic elements. The longest delay on a combinatorial path[10] determines the maximum clock frequency of the resulting FPGA circuit. This is why it is highly desirable to have simple and short combinatorial logic in order to allow fast clock rates.

Most FPGA designs have at least one clock requirement, i.e. they have to work at a certain clock frequency. When the design is very complex and the FPGA utilization is close to 100%, then the fitter has difficulty optimizing the circuit so that it works at the desired clock rate. In general, when the fitter fails to meet the clock requirements, the only solution is to modify and optimize the original design or to reduce its complexity.

3.2.1.2.2 The GETB controller In order to select an FPGA for the GETB, we had to take into account the following factors:

The technology Depending on the way the LEs and the interconnections are configured, FPGAs can be built with SRAM cells, with flash cells or with anti-fuses. For projects requiring complex logic, the SRAM-based FPGAs are used because they support high logic densities.

The speed grade Within the same family of devices, some models are rated at higher frequencies than others. The speed grade gives the delay between neighboring logic cells – the lowest the grade, the higher the maximum clock frequency. Common values are 5 to 7 ns.

Number of pins The number of pins (the package) is extremely important. The same type of FPGA is available in different packages with more or less input/output pins (and a corresponding higher or lower cost). The pin-count must be sufficient to accommodate all connections to the other board components.

Logic elements The number of logic elements determines the complexity of the final firmware and how much functionality it can have. This is the most important factor when selecting an FPGA.

For the GETB project we selected an *Altera Stratix* FPGA manufactured with the SRAM technology. SRAM-based devices can store their configuration as long as they are connected to a power supply. To come back online after a power-off, they need to be reprogrammed – usually this is done using a Flash memory module which stores the device configuration and sends it to the FPGA immediately after power-on.

[10]The part of the *sequential* circuit between two storage elements is called *combinatorial*. At each clock cycle, the data in one set of registers traverses the combinatorial logic and the results enter in the next level of registers. The clock period cannot be shorter than the longest delay in the combinatorial parts of the circuit.

The number of logic elements in the device was chosen based on our estimations about the amount of logic required to implement the various functional parts. As we shall see in the subsequent sections, we decided to integrate into the FPGA the logic for driving the other hardware components (to reduce the number of discrete parts). Our final choice will be presented in Section 3.2.2 after we introduce the internal blocks of the GETB FPGA.

3.2.1.3 PCI interface

The PCI interface had two functions: one was to supply power and the other was to allow the host computer to communicate with the GETB card. For this to work, a PCI interface handler was needed.

Instead of having a discrete component dedicated for the PCI communication, we decided to embed this functionality inside the FPGA. To reduce the development time, CERN acquired a commercial VHDL[11] library (PLD Applications PCI Core, [29]) which implements the PCI protocol. Such a library is commonly referred to as an *Intellectual Property (IP) Core*. The components from the IP core are linked to the rest of the FPGA firmware.

By using a PCI core, the board layout is simplified, we have fewer hardware components and the firmware can be tightly coupled with the PCI. There are, however, two issues:

Voltage requirement When the PCI interface is embedded, the FPGA must be connected directly to the PCI connector. The PCI standard supports two voltages: 5V (the most common) and 3.3V (rarely used). The PCI connectors for the two voltages are different. The Altera Stratix FPGA we've chosen works only at 3.3V so we had to build the GETB card with this voltage as a reference. We discovered too late that there were no industrial PCs with more than six 3.3V PCI slots. Because of this, the final (full) GETB system (which had 60 cards) was occupying 14 computers instead of the 4 or 5 initially planned.

Computer boot-up phase When the computer boots, it scans the PCI bus in order to detect and initialize all devices. It is important that the PCI IP core from the firmware is ready to run by that time, otherwise the computer might hang or might not see the GETB card at all. In the case of the GETB, this problem did not appear – the card was ready[12] before the computer started the PCI scanning.

When dimensioning the FPGA, we had to take into account the number of logic elements used by the PCI core – the estimation from the datasheet was 900 LEs. We should point out that the datasheet numbers are not always accurate – depending on the rest of the firmware and of the clock requirements, the compiler (fitter) might remove parts of the logic which are not necessary or might duplicate some logic in order to improve the timing.

[11] Very high speed integrated circuit Hardware Description Language.

[12] When the computer boots for the first time, the FPGA has to read the firmware from the Flash memory. When the transfer is finished, the firmware is started and the PCI IP core is ready to interact with the PCI bus.

3.2.1.4 Ethernet interfaces

The GETB was designed as a platform for Ethernet applications. To connect a device to an Ethernet network, the following three components are mandatory:

The connector The decision for the GETB was to have RJ45 connectors for copper media. These are smaller, cheaper and more reliable than the equivalent optical fiber connectors. As a compromise between board space and port density, the GETB was built with two Ethernet ports. A third RJ45 connector was installed, but it was not foreseen to be used for Ethernet signals. It had to be used for a proprietary communication protocol between the GETB cards (fast messaging and clock synchronization).

The PHY The PHY is an integrated circuit (IC) which handles the modulation and demodulation of signals on the physical lines. It is a mixed-signal device (both analog and digital). The Ethernet PHY has the RJ45 connector on one side and the Medium Access Control layer (MAC) on the other side. The PHY receives data from the MAC, encodes it and sends it on the line. Each Ethernet connector must have one associated PHY.

The MAC The Media Access Control layer handles the framing of data, the point-to-point transmission and the error detection. There are ICs which integrate both the MAC and PHY functions. The MAC receives sequences of bytes from the user application. It assumes that the byte stream is formatted according to the Ethernet frame format specification (Figure B.5 on page 177).

While the PHY functionality has to be implemented in a dedicated chip (because of the analog interface), for the MAC we were faced with two options. We could either use a discrete component or we could implement the MAC inside the FPGA. One of the main requirements was for a MAC that would allow precise control of the transmission and that would deliver the packets as soon as they arrived (for accurate latency measurements).

We looked first on the market for circuits implementing the Ethernet controller functionality. The available options were reduced to two chips: one was the Intel IXF1002 MAC controller [30] and the other one was the LSI 8101 [31]. The Intel chip included MACs for two Gigabit Ethernet ports, but the two channels were sharing the bus to the upstream device (the FPGA in our case). The interface to the Intel MAC was a proprietary one, mainly designed to be used with the Intel IXP1200 network processor. The second chip, the LSI 8101 had only one MAC interface per device. Both these controllers were designed for consumer applications: network cards or packet switching applications. They were not built for fast packet transmission and reception.

Our second option was the use of a "software" MAC implemented in the FPGA. The functionality of the MAC is described in an HDL dialect and then compiled for the target FPGA. In this way, a fraction of the FPGA becomes the MAC and allows the FPGA to be connected directly to the PHY. The use of a MAC IP core has the following advantages:

- An IP core has almost no lead time, i.e. the component is available immediately (usually as an Internet download). The Intel and LSI chips we mentioned above had lead times of the order of several weeks.

- An IP core comes packaged as a set of HDL files (usually encrypted VHDL). These can be easily integrated into a simulation, which allows the user to get acquainted with the IP core and then develop the FPGA firmware before the hardware is available.
- The complexity of the board is reduced. Integrating a MAC IP core in the FPGA decreases the number of traces and the number of chips on the card, simplifying the board layout and reducing the design costs. This makes the product more reliable and less prone to errors in fabrication.
- An IP core allows for tight integration with the rest of the FPGA firmware. More precise control on the packet transmission and reception is possible.

The main disadvantage of an IP core is that it uses logic resources in the FPGA. By dimensioning the FPGA appropriately, this effect can be minimized.

For the GETB we finally decided to use a MAC core from MoreThanIP [32]. The MoreThanIP MAC core supports 10/100/1000 Mbit/s Ethernet and provides a simple, FIFO-based[13], interface to the client code. For our project we needed to create two instances of the core because we had two Ethernet ports. The logic usage requirements for the MAC were significant – the datasheet specified 3600 logic elements, so in our case we had to provision approximatively 7200 logic elements in the FPGA. This part of the logic had also the highest requirements in terms of clock frequency: for Gigabit Ethernet, data from the MAC to the PHY is sent at 125 MHz (8 bytes per clock cycle).

3.2.1.5 Memory

As we shall see in Section 3.3.1.1, the GETB network tester generates traffic based on a predefined pattern. In order to make the traffic pattern description available to the FPGA, we had to provide on-board memory[14]. We had two types of memory requirements:

1. For the traffic description: large capacity and fast sequential access. Besides the tester, the GETB had to be used for a Network Emulator (Section 3.5.2) – for this application we had to store complete packets in memory. We had to take into account that a fully occupied Gigabit line transfers 125 Mbyte/s (in each direction).
2. For histograms and counter arrays we needed a memory that would permit fast read-modify-write operations and fast random access.

For the first requirement, the most appropriate choice was Dynamic RAM – high capacity DRAM modules are available at low prices; DRAM also supports fast block operations (read/writes). Keeping a balance between board space, component availability, capacity and price, we decided to have 64 Mbyte of Synchronous DRAM for each Ethernet port. We used Single Data Rate (SDR) SDRAM memory chips.

[13] First-In First-Out queue.
[14] The GETB was not supposed to use the memory of the host computer.

While for the second requirement we could have used the on-chip Static RAM that is available in the Stratix FPGAs (2 Mbit), the decision was to install external Synchronous Static RAM (SSRAM), 512 Kbyte for each port. In this way, the on-chip memory was free for other tasks which were depending on very small access times.

3.2.1.5.1 Memory controllers The SDRAM memory blocks were connected directly to the FPGA. The firmware in the FPGA had to take care of the periodic refresh (needed to preserve the contents of the SDRAM) and to handle the column/row-based addressing which is specific to DRAM memories. As a memory controller we used an IP core from Altera [33]. We estimated that two instances of this controller would take 600 logic elements in the FPGA.

The SSRAM memory is easier to work with: it does not need a refresh cycle and is organized as a vector (single address used as an index), not as a matrix (row-column addressing). This makes the controller much simpler in comparison to the SDRAM. We implemented our own controller for the SSRAM.

3.2.1.6 Configuration, debugging and global clocking

3.2.1.6.1 Configuration An SRAM-based FPGA can be configured either from the Flash memory, either using a special cable connected to the JTAG chain[15]. For initial tests and for writing data to the Flash memory, a JTAG connector had to be installed on the card. A computer can connect to JTAG using a special cable[16]. This cable is used to configure the FPGA or write the Flash memory.

3.2.1.6.2 Debugging For diagnostics purposes we installed the following components on the GETB card:

- A 16-pin test header. This connector was supposed to be used with a logic analyzer. Any internal signal from the FPGA could be routed to the test header. In the final version of the hardware we placed jumpers on these connectors and assigned unique identifiers to the GETB cards. At initialization, each card read the status of the test header and reported the 16-bit number.
- Five LEDs. These LEDs were connected directly to the FPGA. When the FPGA firmware was still under development, the LEDs were used to present state information (for debugging).
- A reset button. This button provided a "software" reset for the firmware. It affected only the user code, not the extra controllers running in the FPGA (for the PCI, MAC and SDRAM).

[15] JTAG or the IEEE 1149.1 is a standard for testing circuit boards.

[16] The Altera ByteBlaster cable makes a connection between the parallel port of the computer and the JTAG connector on the card.

3.2.1.6.3 Global clocking In order to measure precisely the delays between two points in a network, the GETB needed a global clocking mechanism. A GPS-based solution was proposed[17]. This required a GPS receiver which would distribute a common clock signal to all the GETB cards on site. To receive the GPS signals, an RJ45 connector was provided on the card. The connector could also be used to send trigger messages from one "master" card to all the others. A GPS fan-out box had to be built to distribute the clock signals.

As the GPS fan-out could not be constructed in time for the ATLAS switch testing program, an alternate solution was found. We used the PCI clock as a time reference. In this way we were able to synchronize the clocks for the cards installed inside the same computer (sharing the PCI bus).

The clocks are synchronized as follows: when the computer reboots, the PCI controller sends a reset signal on the PCI bus. The GETB cards simultaneously detect this reset pulse and they initialize internal time counters. After that, all the cards increment their time counters at the same rate, as given by the PCI clock. In order to re-synchronize the cards (after a firmware upgrade, for example) the host computer has to be rebooted.

3.2.2 Final hardware design choices

In this section we list the final hardware that was chosen for the GETB. Additional details about the components are available in Appendix B.

As we said before, we opted for an SRAM-based Altera Stratix FPGA. We've chosen the slowest model as this seemed to be fast enough for our applications. In order to choose the size of the FPGA, we took into account the estimated logic usage of the main parts of the firmware. Table 3.1 shows our initial estimation for each of the main functional blocks. The last column contains the effective logic usage in the Network Tester. We observe that in this case the amount of user code required was largely under-estimated. But at the time the hardware design was done, our calculations lead to 13700 LEs for the full firmware. Therefore we selected a device with 25000 LEs considering a safety margin of 47%. After constructing a similar table for the pin requirements, we selected a device with 780 pins (593 I/O pins), the Altera Stratix EP1S25-F780-C7. The final list of components is shown in Table 3.2 and a hardware logical block diagram in Figure 3.2.

Function	Estimated LEs	Real LEs
PCI Controller	900 (3.5%)	1061 (4.3%)
2 x SDRAM Controller	600 (2.3%)	690 (2.7%)
2 x Ethernet MAC	7200 (28%)	8240 (32.1%)
Infrastructure	0 (0%)	29 (0.1%)
User code	5000 (19.5%)	13970 (54.5%)
LEs used (out of 25660)	13700 (53%)	23990 (93%)

Table 3.1: Logic resource usage in the GETB FPGA: Initial Estimation vs. Final design.

[17]GPS = Global Positioning System. Any GPS receiver synchronizes its internal time with the time received from the GPS satellites. These satellites have atomic clocks on-board. This makes the GPS time reference very accurate and very useful for doing long distance measurements.

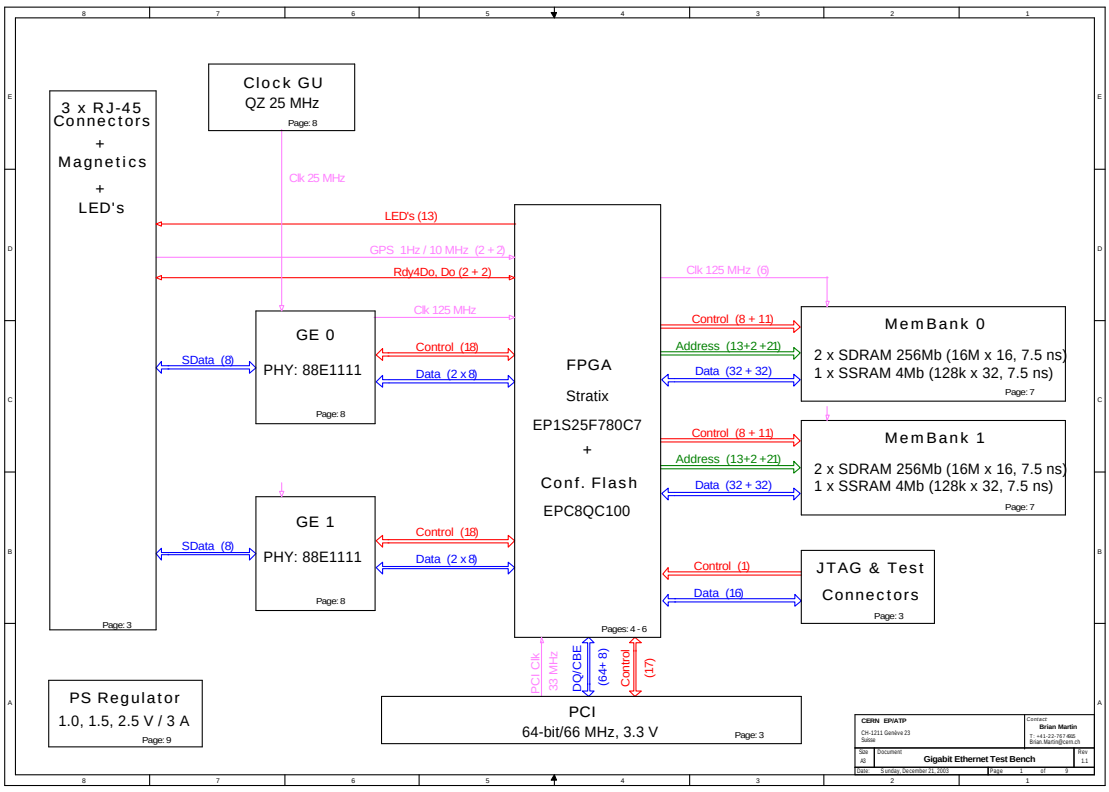


Figure 3.2: Logical block diagram of the GETB card.

3.2.2.1 Schematic, Layout and Fabrication

Once the logical hardware design has been finalized, the GETB schematic and the board layout have been done using OrCAD. Figure 3.3 shows an image of the GETB board with the footprints of the main components (the floor plan). There are also two memory modules on the back side which are not shown.

The board was manufactured by BATM, an Israeli telecommunications company – it was structured on 10 layers and had a thickness of less than 2 mm. There were only 181 components on the board: 145 capacitors, 17 resistors, 13 integrated circuits, 5 connectors and one button. A photo of the finished product is shown in Figure 3.4.

3.2.3 Firmware

The FPGA is the main component of the GETB platform as it controls all the resources of the card. In addition to the user code, the FPGA firmware integrates the controllers for the PCI and Ethernet interfaces. We describe the tools we used for its development, its internal organization and some the techniques we used to optimize it.

Function	Component description
Form factor	Standard PCI card, 3.3V, 32-bit interface to the host.
PCI controller	PLD Applications PCI IP Core (900 LEs).
Connectors	2 x Gigabit RJ45 connectors, 1 x Fast Ethernet RJ45 connector.
Ethernet PHYs	2 x Marvell Alaska 88E1111 Gigabit PHYs.
Ethernet MAC	2 x MoreThanIP Gigabit Ethernet MAC IP Core (7200 LEs).
FPGA	Altera Stratix EP1S25-F780-C7. Speed grade: -7, 25660 logic elements, SRAM-based device, 780-pin Fine-Line, Ball Grid Array package.
Flash memory	Altera EPC8QC100. 8 Mbit flash memory. Memory required to configure the EP1S25 device: 7.89 Mbit.
SDRAM memory	4 x Micron MT48LC16M16A2. Capacity of one module: 16777216 x 16 bit words. Total capacity: 2 x 64 Mbyte
SRAM memory	2 x Cypress CY7C1347B. Capacity of one module: 131072 x 32 bit words. Total capacity: 2 x 512 Kbyte
Diagnostics	One JTAG connector, 5 on-board LEDs, 16-pin test header, 3 test points.

Table 3.2: List of components on the GETB card.

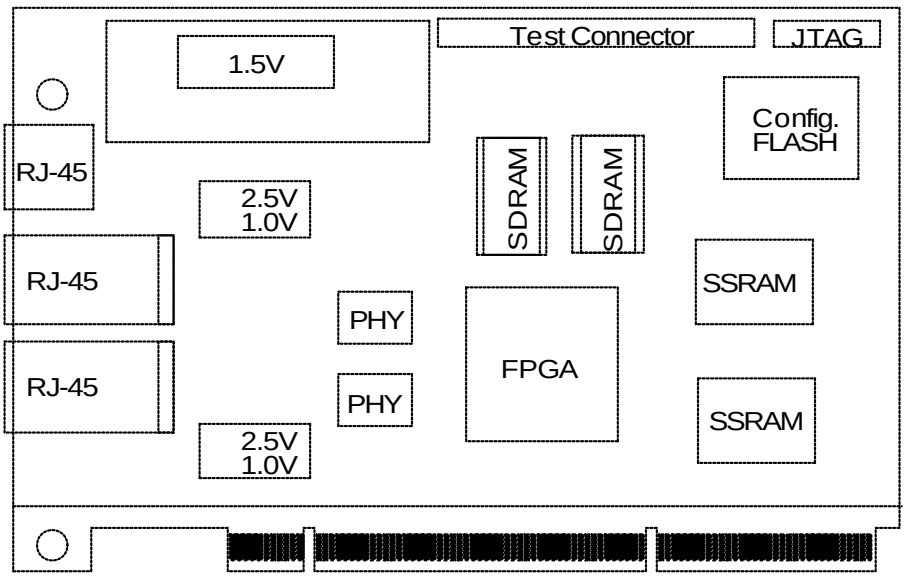


Figure 3.3: The GETB card – Floor plan.

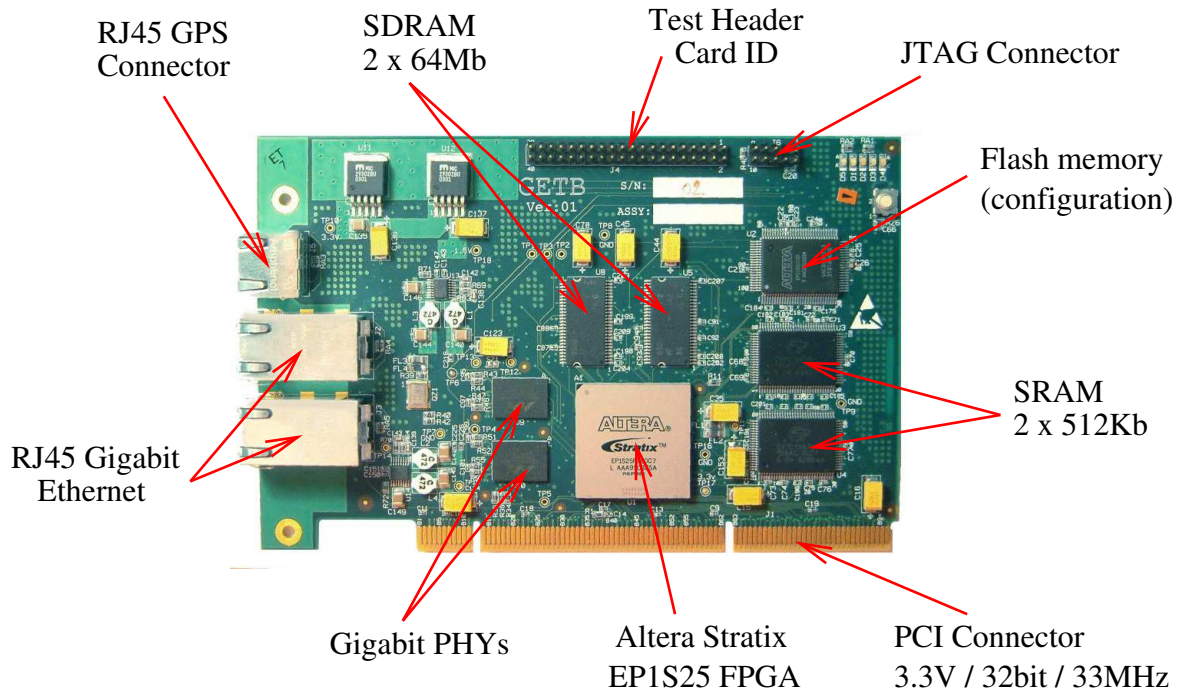


Figure 3.4: The GETB card – Photo.

3.2.3.1 Development environment

The following software tools have been employed for the GETB firmware development.

Design entry – Handel-C The behavior of the circuit in an FPGA is defined graphically using a schematic diagram or, for large designs, textually using an HDL. The most popular languages among hardware engineers are VHDL and Verilog. They can be used to describe any kind of circuit, but they require the designer to think carefully about the gate-level effects of each line of code. This can distract the attention of the developer from the actual algorithm that has to be implemented in hardware.

The current trend in the world of design entry tools is to migrate to languages which allow a more algorithmic approach to the hardware design and leave the low-level details (implementation) to the synthesis tools. We used one of these relatively new languages for the firmware of the GETB platform. Celoxica Handel-C [34] is a language based on ANSI C, which has been augmented with a set of hardware specific constructs – it can be used to describe synchronous digital circuits.

The syntax of Handel-C allows the programmer to define parallel and sequential blocks of code. Each block of parallel code is implemented as an independent circuit in the FPGA. The language has a simple timing model in which each statement is guaranteed to be executed in one clock cycle. Other features include variable bit-widths for integers, synchronization mechanisms between parallel branches of code, the ability to interface to external hardware and the possibility of using chip-specific features (hardware multipliers or on-chip memories). More details about Handel-C are available in Appendix B.5. The

Entity instance (circuit)	Internal ports	External ports	Logic usage
PCI IP core	14	23	4.3%
SDRAM Controllers (2x)	16	18	2.7%
MAC IP cores (2x)	108	20	32.1%
Infrastructure logic	50	26	0.1%
Handel-C code	402	97	54.5%
Top-level entity	-	184 (471 pins)	93%

Table 3.3: Port statistics for the logic circuits inside the GETB FPGA.

output of the Handel-C compiler is a gate-level netlist which becomes the input to the manufacturer-specific synthesis and place-and-route tools.

All the user code for the GETB project has been written in Handel-C.

Fitter – Altera Quartus II The fitter takes as input the definition of a circuit and performs the synthesis, the technology mapping and the final place-and-route for a target FPGA architecture. The outcome is a device configuration file which is used to program the FPGA. Because the "fitting" steps are tightly coupled to a particular FPGA architecture, each manufacturer has its own software suite for this purpose. *Quartus II* is the name of the software package that compiles circuits for Altera FPGAs.

Most FPGA designs are composed of many small circuits which need to be connected together in the final firmware. Quartus allows the user to specify a list of these circuits and how they should be "assembled". This is done via a *top-level hardware description file*.

Then the user needs to specify the pin assignments, i.e. how the internal logic should be connected to the real pins of the FPGA. The list of pins and their functions comes from the board layout/schematic software (OrCAD). In order to do an optimal place-and-route, Quartus needs to know also the clock requirements of the project. As we shall see next, an FPGA may run with multiple clock domains.

The result of the Quartus compilation is a binary file that is used to configure the FPGA. The binary firmware image can also be written to the Flash memory.

The top-level circuit description – VHDL_Gen Within the FPGA, each circuit can be regarded as a black box with a set of inputs and outputs. A complete FPGA design may be composed of tens of individual circuits. These circuit blocks need to be inter-connected between them and some of their ports need to be connected to the I/O pins of the FPGA (so that signals can flow in and out of the chip). All these connections are logically defined in the *top-level HDL file*. The creation of the top-level file can be a tedious task because of the number of inter-connections involved. Table 3.3 contains the main circuits that are instantiated in the GETB FPGA, and the number of I/O ports associated to each of them. We observe that we have a few hundred ports (depending on their bit-width, each port may consist of one or more "wires"). The "external" ports are connected to the pins of the chip; all the other ports are called "internal".

VHDL is generally used to create the top-level file[18]. In VHDL, a connection between two ports is created as follows. First a "signal" is defined (which corresponds to a physical

[18]It is possible to use a graphical block diagram editor or any other HDL.

wire). Then this signal is attached to each of the two ports which are at the end-points of the connection (this mapping is done using "port-maps"). The bit width of the signal must match the widths of the two end-ports. It is assumed that the designer is aware about the direction of each port (in or out) to avoid incompatible connections.

In the case of the GETB project, we understood right from the beginning of the firmware development that the manual creation of the top-level circuit description is a lengthy and error prone process. In order to simplify the development, and to reduce the likelihood of mistakes (which can destroy the FPGA or other board components), we created a tool that automatically generates the top-level description files. This tool (VHDL_Gen, [35]) takes as input the relationships between the different code blocks and generates the required VHDL source code for signals and port maps. In addition to the code generation, VHDL_Gen performs various checks on the design to make sure that the connections are defined correctly (at least from the electrical point of view).

The input to VHDL_Gen uses a compact form to define the relationships between the FPGA blocks. For comparison, for the entire GETB project, the input to VHDL_Gen had approximatively 300 lines, while the generated VHDL top-level description had 2200 lines. The top-level VHDL file contains 24 circuit blocks, 915 ports and 241 signals for the inter-connections.

VHDL_Gen has been written as a module[19] for the Python programming language [22]. The "input" to VHDL_Gen is in fact a small Python program which calls the functions defined in the module. This approach gives more flexibility to the user who has access to all the features of the Python language (in addition to the special code generation functions from VHDL_Gen).

All the VHDL code within the GETB project has been generated using VHDL_Gen. The module was also used to build testbenches for simulation.

Simulation – ModelSim In order to prepare the first versions of the GETB firmware, we used the Mentor Graphics ModelSim gate-level simulation software. We ran simulations for most of the circuits which had to be integrated into the final firmware: the MAC core, the PCI and SDRAM controllers.

Thanks to the simulations we were able to prepare special versions of the firmware before the arrival of the first GETB prototype boards. We used these versions to test the different hardware components on the GETB cards. Having them ready in advance, we were able to check and solve in only one week the minor faults observed on the first GETB prototypes. The ModelSim simulator was no longer used when the final hardware became available.

Device programming The Quartus software produces a device programming file, which contains the configuration of every logic element and every connection in the FPGA. This file can be written to the card over the JTAG connector.

This method, however, does not scale when a firmware modification is required on more than 2 or 3 GETB cards. As a solution, we used the PCI bus to access the JTAG chain and write firmware images to the Flash memory. We customized the Altera JAM Player software [36] for this purpose.

[19] Module = Library of functions.

Because the PCI core was running inside the FPGA, we were bypassing the FPGA and we were directing the configuration bit-stream to the Flash. This process was happening in parallel on all the 60 cards. After the Flash had been written, a special command to the FPGA was forcing a reconfiguration. The entire process took less than five minutes (for all cards).

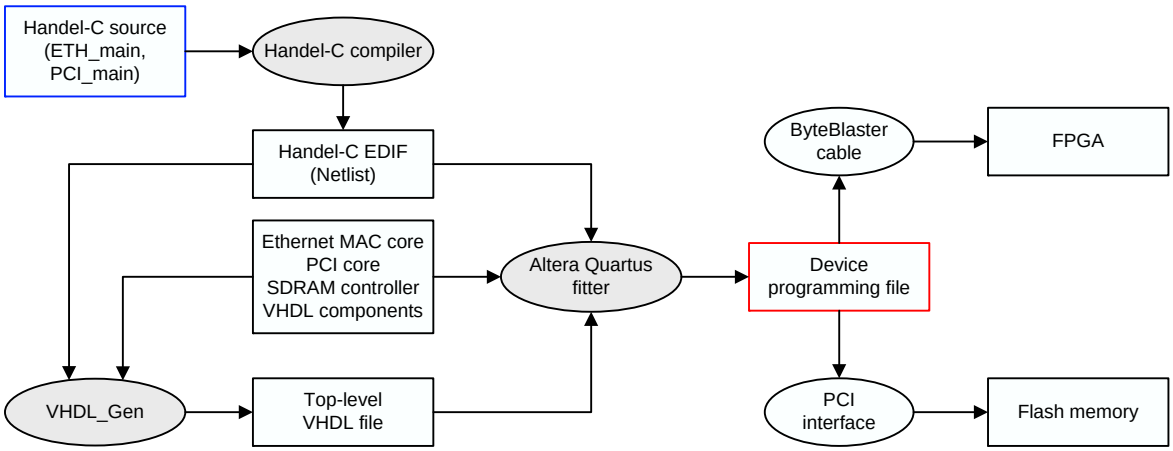


Figure 3.5: The GETB development work-flow.

The tools mentioned above had to be called in a certain sequence and each one had its own set of dependencies. In addition, some of them ran on Windows, while others on the Linux operating system. We used the "make" utility to automatize the build process – in this way, the entire work flow could be started with a single command. In addition, a set of Python scripts were doing automatic verifications over the log messages produced by Handel-C and Quartus – the user was notified in case of severe warnings or errors (some of the warning messages were normal). The running time of the full tool chain in the case of the GETB Network Tester was two to three hours (70% of the time in the Quartus fitter). Figure 3.5 depicts the GETB development work-flow with all the steps and the relationships between them – the next section has details about the inner components of the firmware.

3.2.3.2 Firmware organization

The GETB FPGA integrates the parts listed below – a block diagram of the firmware is shown in Figure 3.6.

PCI IP core This is the part which handles the PCI communication. On one side, the IP core is connected to the FPGA pins, which are then tied to the PCI connector. On the other side, the IP core is connected to a Handel-C code block (PCI_main, see below). The PCI core receives a 33 MHz clock signal from the PCI bus.

Handel-C code / PCI_main This part of the Handel-C code is connected to the PCI IP core. It tracks the PCI protocol state machine and, when it detects a read or write access, it

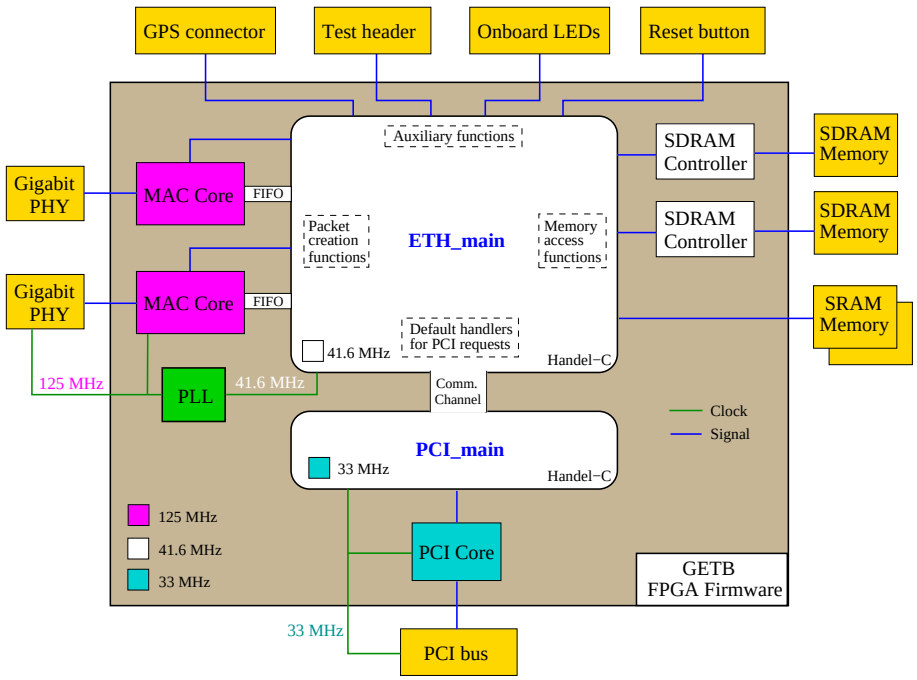


Figure 3.6: Firmware organization inside the GETB FPGA.

issues the appropriate command to the other Handel-C block (ETH_main) which has full control over the rest of the hardware. For read accesses, PCI_main waits for the other block to reply and then it sends the data back to the PCI core and then to the PCI bus. The PCI_main Handel-C code operates synchronously to the PCI, at 33 MHz.

Handel-C code / ETH_main This block represents the main part of the GETB firmware. It contains both the infrastructure code and place-holders for the application code. The infrastructure code is composed of functions for reading/writing the memories, macros to simplify the creation of Ethernet packets, PCI handlers for accessing the memories and configuring the MAC interfaces and so on.

ETH_main provides a framework for the GETB firmware development. The clock frequency at which it operates depends on the application – for the GETB tester, it is 41.6 MHz, for the other projects it is 50 MHz. The ETH_main block has connections to all the other components: the MAC IP cores, the SDRAM controllers, the SRAM memories, the onboard LEDs, the test header, the GPS port, etc.

The configuration of the firmware is done through the PCI bus. ETH_main has a small module which listens to commands coming from the PCI. Some commands trigger actions, others write registers or data into memories and others read the status of the firmware or addresses from the memories. The communication between PCI_main and ETH_main is done using Handel-C channels (for commands) and dual-port memories (for data).

MAC IP cores The GETB contains two instances of the MAC IP core. Each instance is connected to the FPGA pins that go the PHY chips and to the Handel-C user code which generates and receives the Ethernet packets (ETH_main, previously described). The MAC

cores have the highest clock requirements – they receive an 125 MHz clock from one of the PHY chips. The frequency of 125 MHz is imposed by the Gigabit Ethernet standard. The communication between the MACs and ETH_main takes place over a set of Asynchronous FIFOs which can transfer data reliably across the clock domains (125 MHz and 41.6 MHz).

SDRAM controllers The FPGA contains two SDRAM controller instances, one for each memory bank. These controllers connect to the Handel-C blocks and to the FPGA pins. By default, the controllers operate at the speed of the ETH_main Handel-C block. For faster memory access, it is possible to run the controllers at a higher frequency.

Infrastructure logic A set of multiplexers are used to connect the Ethernet MACs to the PHYs such that the MAC will operate correctly for Fast Ethernet (100 Mbit/s), as well as for Gigabit Ethernet. Other components include a PLL[20] used for internal clock generation and counters for the GPS signals.

All the blocks that were described and which are shown in Figure 3.6 are interconnected using VHDL_Gen. The amount of logic used by each block is visible on the last column of Table 3.1 (page 47).

3.2.3.3 Code features and optimizations

The GETB platform packs all the functionality inside the FPGA. As almost half of it is occupied by drivers for the external interfaces (PCI, MAC, SDRAM), the user code has to be carefully tuned to fit into the remaining logic areas and still run at the desired clock rates. For the GETB Network Tester, this goal was more difficult to attain because the tester was supposed to provide identical functionality for the two Ethernet ports. Using a single processing core for both ports was not possible because they would not run at their maximum speed (without increasing the clock frequency). Therefore the only solution was to duplicate all the transmission and reception code, so that each port could work at Gigabit speed and still let the system run a reasonable clock rate.

We realized the limitations in clock frequency as soon as we started the development on the GETB hardware. While the FPGA had been advertised to support speeds of hundreds of MHz, we discovered that when the logic utilization of the chip increases above 60%, the maximum clock frequency drops down to a few tens of MHz. The MAC core outputs one byte per 125 MHz clock cycle. The interface between the MAC and the Handel-C code (ETH_main) has a width of 32 bits. Therefore, in order to be able to send and receive at line rate, the minimum frequency for the user code is $\frac{125}{4} = 31.25$ MHz. Of course, the frequency needs to be higher than that in order to allow time for packet processing.

One major constraint in this FPGA design was the MAC which had very strict and very high clock frequency requirements. The PCI core had also a very strict, but a much smaller requirement: 33 MHz. The only clock that allowed some flexibility was the clock driving the Handel-C code. After enabling all the optimizations in the Altera Quartus software we were able

[20]PLL = Phase-locked loop. A circuit that uses a reference clock and can synthesize clock signals of different frequencies by division and multiplication.

to run the GETB Network Tester code at 41.6 MHz (with an FPGA utilization of over 90%). The other projects based on the GETB used fewer logic elements and so they were able to run at 50 MHz.

In addition to the automatic optimizations done by the Quartus fitter and the Handel-C compiler, we performed a number of manual improvements at the level of the user code (ETH_main) in order to increase the clock frequency and to reduce the logic usage:

Parallelizing code blocks By having more than one task in parallel, the circuit does more work per clock cycle. Handel-C has built-in support for code-level parallelism. It allows the programmer to specify which code blocks run in parallel and which run sequentially. Parallelism is often used in conjunction with *pipelining*. An operation is divided into smaller sub-tasks and each task is handled by a parallel process. In an FPGA, each "process" becomes essentially a logic circuit. Within the GETB, for each Ethernet port, the transmission and reception are handled by parallel processes. Within each process there are many small parallel pipelines for the operations that allow this kind of processing (for example the analysis of the packet headers). Code parallelism improves the clock frequency, but sometimes increases the logic usage.

Code replication A sequential segment of Handel-C code can handle one data item at a time. When this segment is inside a loop and it has to be executed for many iterations, it can be useful to have several "copies" of the code and each copy to process a different data set. Handel-C has a feature which allows the user to easily generate replicas of code segments. For example, the iterations of a "for" loop can all be executed in parallel, or they can be assigned to a number of "threads" that run in parallel (each thread having a small number of iterations assigned to it). This feature is controlled by compile-time constants. It allows the developer to tune the code and make a trade-off between speed (many parallel branches) or space (less branches, more processing time). This feature has been used inside the GETB to distribute received packet data to several classification engines (for making histograms). Each "engine" was checking if the packet matched a certain condition (coming from a certain source, having a certain tag, etc.)

Code sharing Some operations which may seem very simple, can use a lot of resources in the hardware. Examples are adders, comparators or multiplicators. The amount of logic consumed depends on the width of the operands. For the GETB, we used a a feature of Handel-C which allows a piece of hardware to be shared by multiple parallel or sequential processes. For concurrent usage of the shared hardware, the user is responsible for the arbitration.

Using on-chip memory instead of registers Modern FPGAs contain a few megabits of on-chip SRAM memory. This memory has very fast access times and it supports dual-port read and writes[21]. Handel-C has native support for this hardware feature and it allows variables to be stored either in registers, either in the on-chip memory. Within the GETB, we tried to use as much as possible the on-chip RAM in order to save registers (which are implemented in the logic elements). However, doing so affects the performance because certain operations

[21]Data can be written to an address, while another address is being read.

need to be serialized. For example a simple incrementation of the contents of a memory cell requires two clock cycles: one to read the data and one to write the updated value. With registers, this operation can be done in one cycle.

Minimizing the bit-width of variables Throughout the code we tried to use the minimum number of bits for the variables stored in registers (each register consumes one logic element). Where possible, the registers have been reused in different parts of the code (this, however, creates another problem because it requires multiplexers in the hardware).

Avoiding comparisons For the conditional blocks (*if* or *while* statements) we tried to avoid using "less-than" or 'greater-than" comparisons because they require implementing the hardware to compute the difference. Where possible, we replaced the comparisons with the "equal-to" or "different-from" operators – their implementation uses less logic.

Minimizing the complexity of the combinatorial logic Handel-C is a cycle accurate language: any assignment statement takes one clock cycle to execute. This means that the compiler will generate all the hardware required to compute the assigned value in one clock cycle. For complex expressions, this leads to heavy combinatorial logic which implies long delays and consequently a reduced clock frequency. To improve the performance, it is sometimes necessary to break complicated expressions and execute the parts as multiple statements. This does not always slow down the code because the operations can be pipelined.

Memory access The GETB card has three types of memory: the on-chip SRAM in the FPGA, external SRAM and external SDRAM. The on-chip SRAM has single cycle read/write access times and is directly supported by Handel-C. The external memories need interface code to operate. The external SRAM is controlled by Handel-C and proved to be fast enough for our purposes (read-modify-write operations for histograms). For the external SDRAM we had to interface Handel-C to the SDRAM controller. For the Network Tester we used the SDRAM *block read* functions, reading chunks of eight 32-bit words – this proved to be enough for sending data at Gigabit line speed.

In the tester, each Ethernet port had an associated memory bank. Another project built using the GETB, the Network Emulator (Section 3.5.2) needed faster access times. The problem was solved by allowing the memory to run with a faster clock and using both memory banks in parallel (extending the word width from 32 to 64 bits). A small part of the Handel-C code was acting as a memory server, receiving and delivering data from the memory via a set of asynchronous FIFOs.

Crossing clock domains Transferring data across two different clock domains requires a handshaking (synchronization) protocol in order to protect from the effects of metastability. This process takes time, this is why cross-domain transfers are generally avoided. Within the GETB we tried to minimize the number of different clock domains by running the memories at the same clock frequency as the main part of the Handel-C code.

For the link between ETH_main and PCI_main we used cross-domain Handel-C channels – a data transfer through such a channel takes 4 clock cycles (instead of one, when the channel end-points are in the same domain). In Handel-C, channels are used to signal conditions (see the next paragraphs). For passing data, we used dual-port memories –

their access ports can run at different clock speeds. The same dual-port memories can be used as queues (FIFOs) – these were used for the communication with the Gigabit MAC core.

We explained in Paragraph 3.2.1.6.3 (page 47) that the GETB cards have their clocks synchronized over the PCI bus. The counter for time works at the PCI clock frequency (PCI_main, 33 MHz). This time counter is used for latency measurements inside the main part of the firmware, in ETH_main (41.6 MHz). We used a dual-port memory to send the value of this counter between the two domains. In order to avoid any data corruption problems (which would lead to incorrect measurements), we used Gray encoding[22] for the time values.

The Handel-C code for the Network Tester contains 14 parallel "processes" which run continuously. Each of these processes "spawns" a few other smaller threads for doing simpler jobs. To coordinate and synchronize all these activities, we used the following features of Handel-C:

Semaphores (Mutexes) These are standard synchronization primitives found also in software programs. They are used to protect a shared resource from concurrent access (mutex = mutual exclusion).

Channels These are structures which appeared for the first time in the Occam language [37] for transputer microprocessors. Channels are used to synchronize two parallel blocks of code. A channel can be read or written. For either of these operations to succeed, there must be a writer or a listener on the other side of the channel. When a process tries to write to the channel and the other process is not ready to read yet, the writer will block and wait. The same rule applies if the reader is the first who tries to use the channel. Channels can be used within the same clock domain (between the processes running in ETH_main) or between two different domains (between ETH_main and PCI_main).

Many of the optimizations we performed implied a compromise between speed and logic usage. In the final versions of the Network Tester, it was no longer possible to fit the entire design in the FPGA, if all the features were enabled. This is why, we've made use of conditional defines throughout the firmware source code. It allowed us to quickly add or remove parts of the functionality or to compile firmware versions optimized for speed (with less features) or for functionality (more capabilities but slower operating frequency).

3.2.3.4 Testing

Firmware development for FPGAs can be difficult because of the lack of the standard diagnostic tools available in software: on-line debuggers, *print* statements, breakpoints, watches and so on. The GETB provides several features for troubleshooting, as described in Section 3.2.1.6. We used them for the initial firmware versions, when the code was not mature enough and the

[22] The *reflected binary code*, also known as Gray code, is a binary numeral system where two successive values differ in only one digit. In this case, a bit error in the encoded value would change the decoded decimal value by only one unit.

advanced features were not available. The first hardware prototypes have been checked with these special versions. These versions (described next) have been produced before the hardware became available and they were aimed at validating specific parts of the hardware. They were developed and debugged using the ModelSim simulator (Section 3.2.3.1). When the GETB prototypes arrived, these firmware images were simply used to program the FPGA and check the behavior of the board.

Electrical wiring This version of the firmware contained almost no code. All the output pins had their default values (chosen for minimal impact in case of a wiring mistake). This version was supposed to detect any short circuits on the connections involving the FPGA.

Handel-C user code This firmware contained a small Handel-C program which was outputting values to the on-board LEDs, the Test Header and the GPS connector. With this code we validated all the available hardware debugging methods.

Memories We had two firmware images (SRAM, DRAM) for testing the memories using Handel-C programs. The code was writing and then reading data from the memories – in case of mismatch, the on-board LEDs started blinking and the test would stop.

PCI interface The first test for the PCI interface was very simple and involved only the instantiation of the PCI IP core in the FPGA. After rebooting the host computer, we would check if the GETB card had been detected properly by the operating system.

MAC interface The most complex test was done for the Ethernet ports. It contained Handel-C code that was creating packets and was pushing them to the MAC (once every second) – this validated the Ethernet transmission. For the reception, we programmed the firmware to listen for packets, read them from the MAC, check their contents and – if a certain pattern was detected – extract a command code and execute it. With this mechanism in place, we were able to define new commands for more advanced tests: configuring the MAC core, sending data over the GPS connector, verifying the PHY chips and so on.

The method described at the last item allowed us to configure the card, monitor its operation and program it to execute certain tasks. Even though the control of the GETB over Ethernet was implemented first, we knew that this was not the long-term solution. In the final GETB firmware, the card had to be managed entirely via PCI, using the Python-based control software which is presented in the next section.

3.2.4 Control software

Modifications to the FPGA firmware are difficult and time consuming. This is why the GETB was programmed to offer "services". A set of relatively simple functions were implemented in the FPGA. These were then used as building blocks for more complex behavior – the hardware functions were accessible to the user through the *GETB control software* (GETB/CS).

The aim of the GETB/CS was to provide a common control infrastructure for all the GETB-derived projects. We needed a system flexible enough to allow complete customization for the

different applications. We also wanted to be able to automatize easily the actions of the GETB cards. This requirement practically excluded the development of a Graphical User Interface specifically for the GETB (which we thought would limit the functionality and would be difficult to maintain and further improve). Finally, our decision was to build almost all the GETB control system using the Python scripting language [22].

Python is a general-purpose programming language, with a syntax similar to "Matlab" or to "C". Like Matlab, Python is an *interpreted* language, i.e. it does not require a *compilation* step like "C" does. This brings many advantages, the most important being the very fast turn-around time. Python allows the user to write programs (scripts), but it also supports an interactive shell where commands (functions) are executed as soon as they are typed – this feature is very useful for testing new code or for learning the functions provided by a module[23]. Python programs can be written using the object-oriented programming paradigm and they have access to the Python standard library which comprises more than 200 modules.

The GETB/CS is divided into two main components: the *server* component – which runs on the computers hosting GETB cards; and the *client* component running on any other machine which needs to access the resources of the servers.

GETB Server software The GETB was developed and was used exclusively with the Linux operating system. In Linux, user programs are not allowed to access directly the hardware – this can only be done through a kernel driver. For our project we used a generic driver which was developed at CERN. *IO_RCC* [38] is a Linux kernel driver which gives access to any hardware device from user space – this tool greatly simplified the management of the GETB cards. Using the IO_RCC driver, we were able to read and write to the memory of the GETB and also to send commands to the FPGA.

We've seen in Section 3.2.3.2 that the main part of the GETB firmware is located in the ETH_main logic block. This code accepts a set of commands that configure or trigger certain actions. The only way the user can issue these commands is via the PCI bus and then traversing the PCI_main logic layer. By means of *memory mapped I/O*, the IO_RCC module enables us to interface to the ETH_main block and to access the GETB hardware.

On top of the services provided by IO_RCC we've built a Python *wrapper library*. This gave us the possibility of controlling the hardware from the Python shell. The GETB server software uses this *wrapper* to manage the GETB cards in one computer. It detects, configures and then constantly monitors the GETB cards. At CERN we have 14 GETB servers; a single server manages between 4 and 6 cards.

The most important task of the server software is to give access to the cards from remote computers running the GETB client software. This is done by exporting a sub-set of the implemented functions and allowing them to be called from remote machines. The technique is called *Remote Procedure Call* and it allows a remote client to execute functions on a server and retrieve the results. There are several protocols which can be used:

[23] A Python module is a collection (library) of functions and classes.

CORBA[24], SOAP[25], Java RMI[26], XML-RPC[27], etc. We've chosen XML-RPC because its use was straightforward and because, after several improvements described next, the attained level of performance was satisfactory for our requirements.

The GETB servers are never contacted directly by the end-users – they are just *service providers*. The users are supposed to interact with the GETB cards only via the GETB clients.

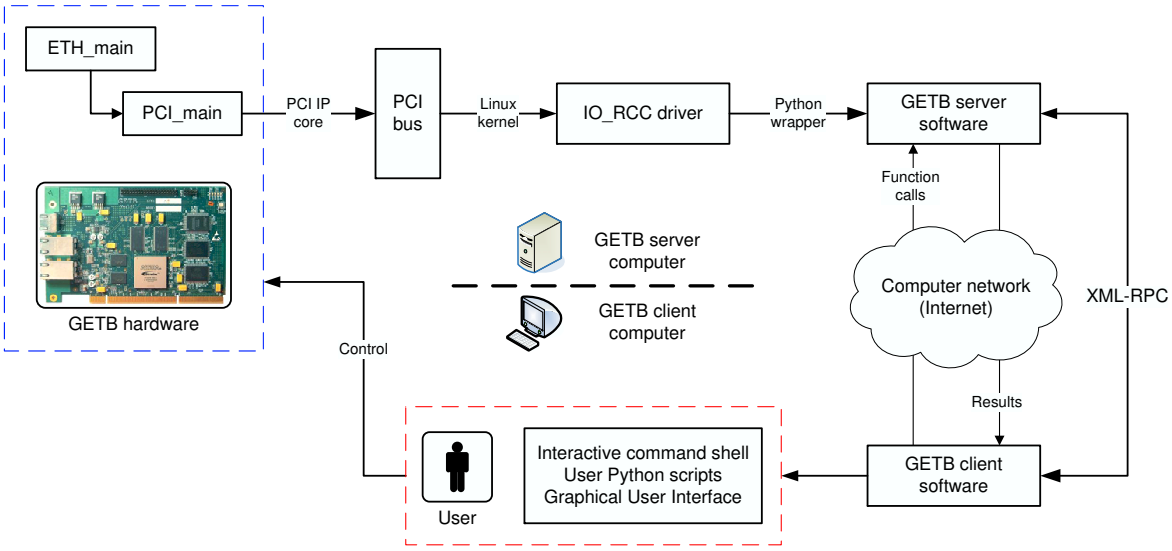


Figure 3.7: The GETB control chain.

GETB Client software The client software is packaged as a library of functions that can be called to perform various tasks using the GETB cards. These functions are used from scripts written by the user. The GETB clients connect to the GETB servers using XML-RPC.

On the GETB client, the two ports of the GETB card are seen as completely independent entities, i.e. the user does not need to be aware of the physical location of a GETB Ethernet port. The client software (library) runs on any computer that can run a Python interpreter. We used the client from Linux, Windows and Mac OS. The GETB client works either as a module (when included in other programs) or as an interactive shell (command-line interface). A Graphical User Interface for displaying statistics in real-time from all the cards is also available.

The method we used to connect the clients to the servers (XML-RPC) is very easy to use, but not very fast. Like most RPC protocols, XML-RPC is sensitive to the network latency and to the time to establish connections between clients and servers. We used the following two techniques to overcome these issues:

[24]Common Object Request Broker Architecture.
[25]Simple Object Access Protocol.
[26]Java Remote Method Invocation.
[27]Remote Procedure Calls using XML, the Extensible Markup Language.

Parallelizing the RPC calls In many cases, the GETB client had to connect to all the servers and execute certain functions. The time spent to process a single call was mostly dominated by the network round-trip time. To overcome this problem we parallelized the calls using threads. For each call, the GETB client spawns a thread; all threads run in parallel and in this way the network latency is hidden. We did not notice any problems when starting and stopping a few tens of simultaneous threads. This technique minimized the effect of the network round-trip time.

Using persistent network connections The second source of delays comes from the time to establish the network connections between the client and the server. The XML-RPC protocol uses HTTP[28] for transport, which in turn is based on TCP[29]. By default, in the standard implementation[30], XML-RPC creates a new TCP connection for each RPC call. Once we understood this, we switched to a different XML-RPC library which was implemented in C and was optimized for speed. That implementation used persistent TCP connections, i.e. it created the connection to a server only once. Details are available at [39].

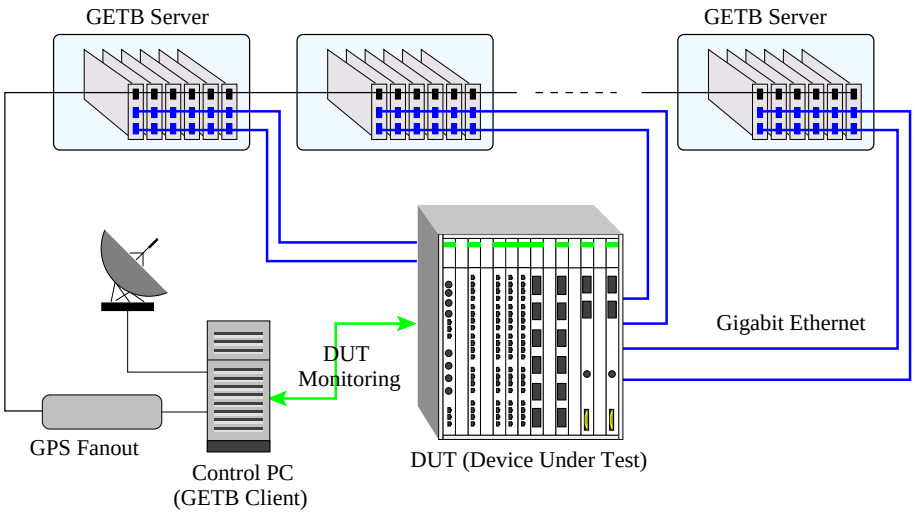


Figure 3.8: Typical testbed setup (block diagram).

Figure 3.7 shows the entire "chain of command" that allows the user to control the GETB hardware. In Figure 3.8 we show the typical setup of the GETB system, comprising a single[31] GETB client (the Control PC) and multiple GETB servers. The figure is relevant for the Network Tester and shows the cards in all the GETB servers being connected to the Device Under Test.

The GETB control software, with its two components, has been successfully used for all the GETB-based projects. The developers of each project used GETB/CS as a basis on which they

[28] Hypertext Transport Protocol.
[29] Transport Control Protocol.
[30] The one which is part of the Python distribution.
[31] The GETB/CS supports any number of clients, but only one should be allowed to make modifications to the cards (configuration). But otherwise, all clients can monitor the cards, read their status and their counters.

added project-specific features. Details about the use of GETB/CS in the Network Tester are given in Section 3.3.3.

3.3 The Network Tester

The main application of the GETB platform is the Network Tester. It was created to allow us to evaluate network devices for ATLAS and identify those which best fulfill our requirements.

3.3.1 Features

The Network Tester uses the GETB card to implement a traffic generator and measurement system. It can send and receive traffic at Gigabit line speed while computing averages in real-time for the most important parameters of a flow (packet loss, latency and throughput). Being an FPGA-based architecture, all processing is done by hardware without any CPU-load on the host system. Two transmission modes are supported: one in which each port is fully independent of the others (Independent Generators, IG mode) and one in which a port transmits only when requested by another port (Client-Server, CS mode).

3.3.1.1 Transmission – Independent generators

In the IG mode each port in the system sends traffic according to a list of packet descriptors loaded into the SDRAM of the card. Each TX[32] descriptor is used to build one packet. The firmware uses the fields in each descriptor to create the outgoing packets. The user can configure the Ethernet and IP headers, the inter-packet gap (IPG) and the packet size for each descriptor independently, allowing a wide range of traffic patterns to be generated. For example, a negative-exponential random number generator used for the IPG produces a Poisson traffic stream. In addition to raw Ethernet and IP packets, special frames like Flow Control and erroneous frames can be transmitted. The set of descriptors is created using the control software (Python) and downloaded through the PCI interface into the SDRAM of the card. The length of this list is limited only by the available memory. The firmware can send a fixed number of packets or can cycle indefinitely through the descriptor list.

3.3.1.2 Transmission – Client-server

The second mode of operation – the client-server (CS) or request-reply mode – emulates the traffic produced by the data-intensive applications in the ATLAS TDAQ network.

The ports of the tester are divided into two sets: clients and servers. The servers send data only in reply to requests coming from the clients. The load on the network is regulated by the clients, which use a *token-based* mechanism for issuing requests.

[32]We shall use the abbreviation TX for Transmission and RX of Reception.

Any client keeps track of the number of requests it has sent (and not yet received an answer). Each un-answered or outstanding request represents a *token*. The client sends requests until it reaches an upper limit for the number of tokens, the *High Watermark*, H_W. After that, it stops sending and waits to get enough answers so the number of tokens decreases below the *Low Watermark* or L_W. Only then it resumes sending requests, again up to H_W. Basically, the client limits the maximum load on the network with H_W and enforces a minimum load with L_W. An in-depth discussion about this kind of traffic shaping can be found in Chapter 4.

The requests are usually small frames (64 bytes) while the replies consist of one or more maximum sized frames (1518 bytes). Congestion is typically created towards the clients (many servers send large amounts of data to a small number of clients). To avoid stalling the system because of packet loss we implemented error recovery mechanism on both ends relying on packet loss detection and timeouts (see also Section 3.3.1.3).

The client-server mode tries to emulate the behavior of the SFI, L2PU and ROS components of the ATLAS TDAQ (Chapter 2). By choosing the appropriate values for the concentration ratio (number of servers vs. clients), the values for the token limits and the number of replies, we can emulate more accurately different traffic scenarios in the ATLAS network.

For example, the SFI has a configurable number of $N = \textit{Outstanding Requests}$. It always make sure that it has N requests in the network, waiting to be served (receive data from the servers, the ROSs). When it gets a reply (event fragment), it immediately issues another request. We can emulate this behavior with the GETB by setting $H_W = N$ and $L_W = N - 1$. The L2PU, on the other hand, sends N data requests, waits to get *all* answers, processes the data and continues by sending another N. This corresponds to $H_W = N$ and $L_W = 0$ in our system.

Both modes of transmission (IG and CS) maintain individual statistics for the number of packets and bytes that are transmitted to each of the other tester ports in the system. All transmitted packets contain information that is used to detect packet loss and to measure latency (see next section).

3.3.1.3 Reception

The packet receive path is responsible for updating statistics and histograms. Each port keeps track of a set of global counters (total number of frames, bytes, different frame types, etc.) and a set of counter arrays which are updated per source-destination flow: packet loss, average latency, average IPG.

The tester detects packet loss in real-time by embedding a sequence number into each packet sent – any gap in the received sequence numbers means that the packets have been either lost or reordered on their way to the destination. The one-way latency is measured by marking each packet with a timestamp (the clocks are synchronized between cards, see Section 3.2.1). While a test is running, the information about packet loss and latency is available in real-time for each source-destination pair.

For an in-depth analysis, the user can define histograms (for latency, IPG, packet size and queue utilization). A set of configurable rules based on source ID or VLAN priority can be used

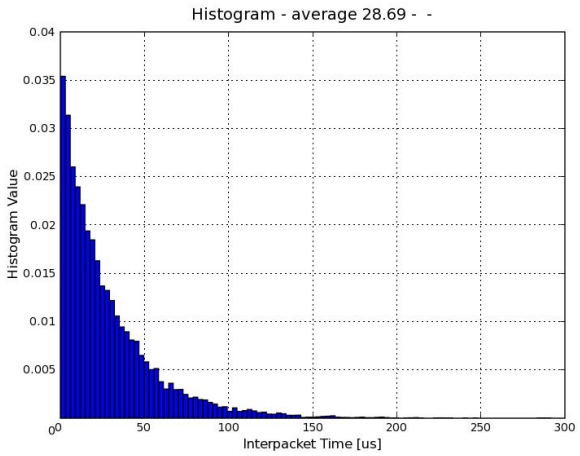


Figure 3.9: Sample histogram – Inter-packet time – Negative exponential distribution.

to filter the frames to be logged in a histogram (conditional matching). For each histogram the user can set the minimum and maximum values, the resolution and the number of bins. As an example, we show in Figure 3.9 the histogram of the inter-packet times for a stream of Poisson traffic with Negative Exponential inter-packet time. In this case the tester was configured to send a stream of packets with size 1518 bytes and an IPG given by a NegExp distribution – the distribution was configured for a load of 30% which corresponds to an average IPG of 28.8 us.

3.3.1.4 Packet capture mode

The receive functions in the GETB support a *packet capture* mode. When this is enabled, the card saves some information about each packet it receives (the equivalent of an RX descriptor). The data is written into memory and can be read at the end of the test. Each capture record contains the latency, the inter-packet time, the time of arrival, the ID of the source port[33] and a few other counters specific to the client-server mode.

The packet capture feature allows us to create more accurate latency histograms and can also offer insight on the traffic patterns. This mode was essential in understanding the ATLAS traffic pattern (more precisely, the analysis from Section 4.4). Basically, we can observe a time series evolution for any of the fields saved in the RX descriptor. The card can save data for 100 seconds for maximum-sized packets (1518 bytes) coming at line-speed (81000 packets/second). The capacity reduces to 5 seconds for small packets [34].

An example is shown in Figure 3.10. We configured two tester ports (A and B) to send traffic at line-speed to a third one (C). This is a case of 2:1 over-subscription. The output queue of the switch port where C is connected rapidly fills up and the switch will start dropping packets. We enabled the capture mode at C and we plotted the latency and the number of lost packets as functions of time. The first plot shows that the latency increases with each new packet, until

[33]The GETB port from where the packet originates.
[34]64 byte packets at the maximum line rate of 1.488 million packets/second.

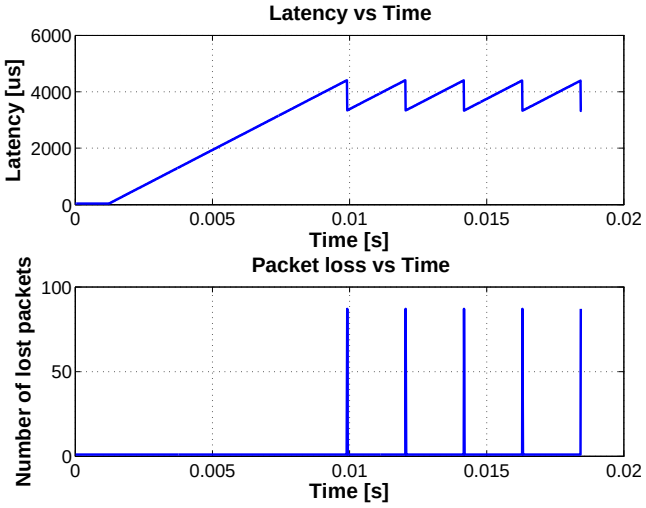


Figure 3.10: Packet capture mode – 2:1 over-subscription – Plots for latency and loss.

the output queue is full and the switch starts losing packets. From the maximum latency we can determine the size of the output queue of the switch. The speed of increase is given by the over-subscription factor. The second plot shows the packet loss. We observe that the switch losses packets in compact bunches of $\approx$ 80 packets[35]. This explains the "saw-tooth" shape of the latency curve. After the switch drops 80 packets, the newly arrived packets will experience a shorter latency, until the queue fills up again (they will have to wait less time).

3.3.2 Implementation

The implementation of the GETB tester had to take into account two hard requirements. Firstly the two Gigabit ports available on the card had to be seen by the user as fully independent entities. And the second was to be able to send and receive packets at line speed for all packet sizes, on both ports simultaneously. One has to make a compromise between the amount of features that are available and the maximum speed (clock frequency) at which the code can run at. As the complexity of the code increases and the FPGA logic resource utilization approaches 100%, the maximum clock frequency decreases dramatically (in some cases the resulting circuit ceases to function correctly). Thanks to the FPGA architecture, we fulfilled the performance requirements by optimizing the code to make use of the parallelism as much as as possible.

The transmission (TX) and reception (RX) of packets is handled by parallel "processes" as it can be seen in Figure 3.11 (each process runs on dedicated hardware and is equivalent to a virtual CPU). For TX there is a process that decodes descriptors from the SDRAM and puts them into a queue. Another process gets data from this queue and programs the Ethernet MAC[36] for the actual packet transmission. Both TX processes are tuned so that wire-speed can be achieved in all conditions. In order to obtain the desired average line rate, the spacing between packets is

[35] We assume this happens for reasons of efficiency.
[36] Details about the MAC interface can be found in Section B.2 at page 175.

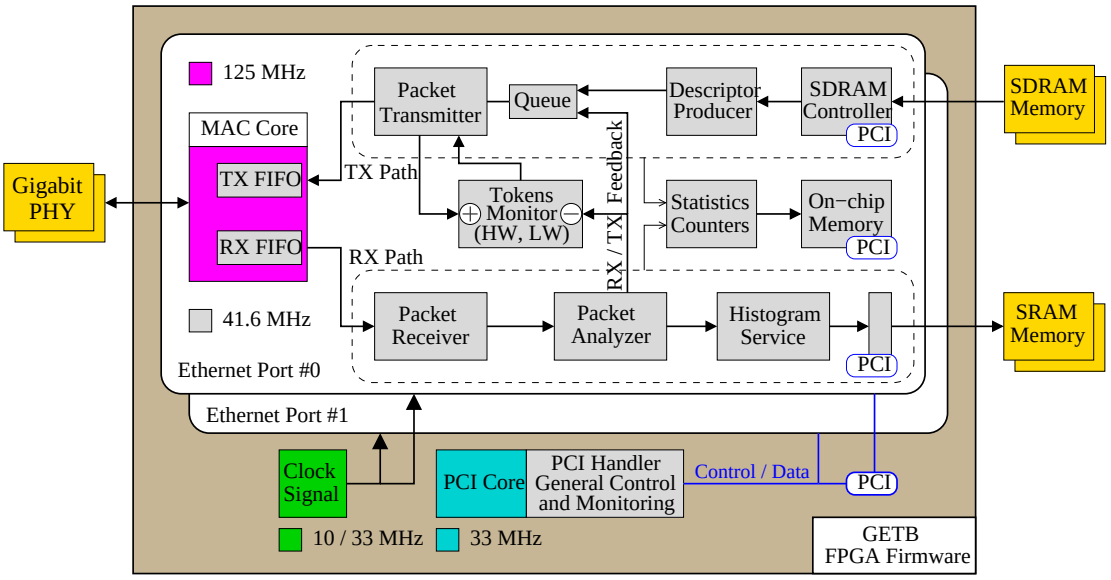


Figure 3.11: The GETB Network Tester – Firmware block diagram.

modified according to the IPG field in the descriptors.

It can be seen in the figure that there is a feedback path from RX to TX which is active in the client-server mode – in this case requests coming from clients are translated into commands that are pushed into the transmission queue. The TX process in the client will send requests as long as it stays within the token limits (Section 3.3.1.2).

The RX path is also served by two processes – one of them (the Packet Receiver) makes sure to take all data available from the MAC core (to avoid any queue overflows inside the MAC) and the other one (Packet Analyzer) tries to process all the incoming packets – it updates counters, histograms and tokens. In the cases when there are too many requests coming or too many histograms that need to be updated then the Packet Analyzer will discard some of the packets.

The behavior of the firmware is slightly modified in the Packet Capture mode. In this mode the TX descriptors are read from the SRAM. The SDRAM is reserved for the captured packets (the RX descriptors).

All the firmware blocks are configured and monitored by the control software via the PCI interface. The external and internal memories[37] can also be read using the host PCI interface.

3.3.3 Operation

The tester operation is entirely managed by the control system described in Section 3.2.4. Python scripts are available to run all the tests described in the ATLAS requirements document for test-

[37]The FPGA has 2 Mbit of internal memory (SRAM). It is used for counters and to implement queues between the various blocks of code. This memory features dual-port access, i.e. different addresses can be read and written simultaneously (even from different clock domains).

ing candidate devices [4]. The tests are performed automatically by iterating over the parameter space (i.e. modifying the network load, traffic pattern, switch settings, etc). The results and the plots are automatically saved. At each test iteration consistency verifications are performed.

In order to control the device under test (DUT) from the testing scripts, we developed a Python interface (called *sw_script*) that is used to configure and monitor the tested device (*sw_script* is presented in Section 5.2). Using this feature, the statistics reported by the tester are cross-checked against those reported by the DUT. This switch-dependent procedure is currently implemented for the products of the major switch manufacturers.

In addition to the collection and the verification of the test results, we've developed Python scripts that produce plots using the results and then group these plots in report templates. These report templates proved to be very useful because they allowed us to quickly form an opinion on the overall performance of a device.

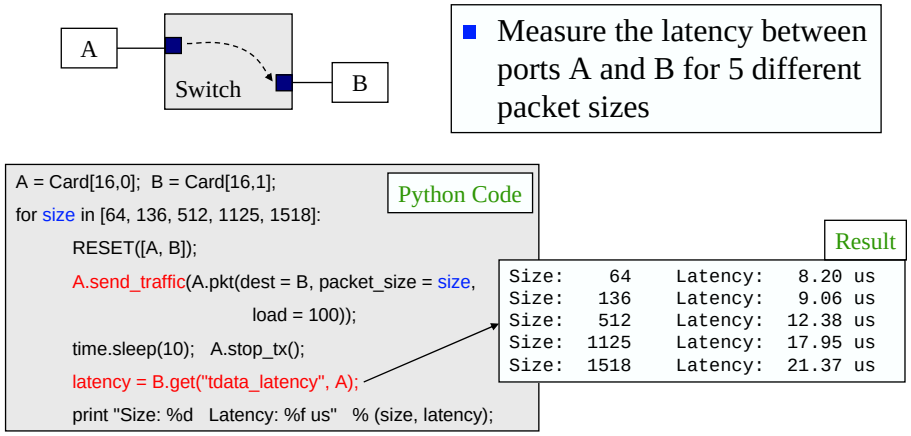


Figure 3.12: Sample Python script – Measuring the latency between two ports.

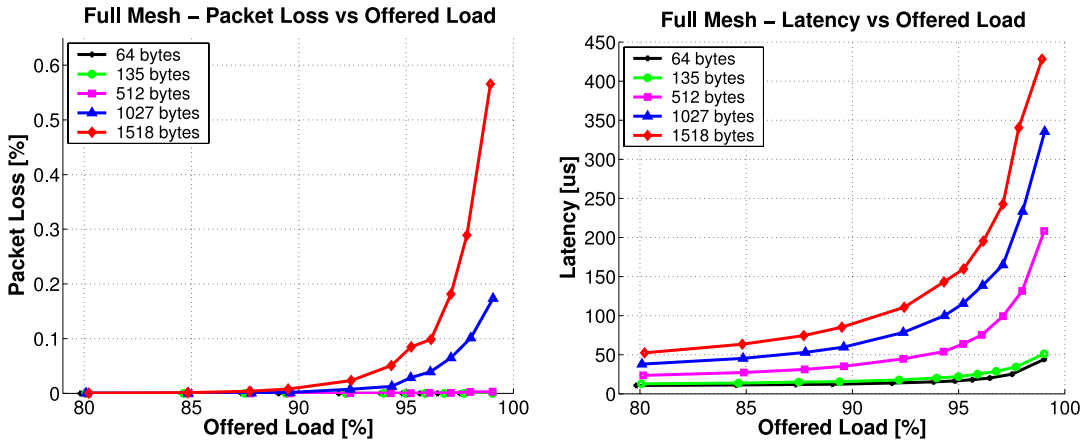
As an example, we show in Figure 3.12 a Python script used to measure the one-way latency between two switch ports. We use ports A and B, which are both on the same GETB card (with index 16). Then we iterate through 5 packet sizes and for each "size" we configure port A to send packets to port B at line-speed. After 10 seconds we print the average latency during the test period.

3.4 Sample results

In this section we present a few results obtained using the GETB Network Tester. These are typical examples of tests we used to run on each candidate device. We show two types of results. Section 3.4.1 presents an example of how we characterize the *switching performance* of a device using a particular class of traffic pattern (full-mesh). Section 3.4.2 contains measurements that characterize one particular *feature* of the device – in this case, the buffering capacity. These tests are part of the standard methodology we defined in [4].

3.4.1 Fully-meshed traffic performance

One of the most common ways to assess the raw performance of a switching device is to study how well it handles fully-meshed traffic. "Fully-meshed" means that each port has to forward packets to all the other ports, usually in a random order (a tester node transmits to all the other nodes). Using this kind of traffic one can determine the limits of the switching capacity of the device. In Figure 3.13(a) we present the packet loss rate measured as a function of the increasing offered load. The device under test does not lose anything for small packet sizes, but drops up to 0.6% of the traffic when we use large packet sizes. In Figure 3.13(b) we plotted the average one-way latency measured in the same conditions. As the offered load is increased and because the switch needs more time to forward all packets, it stores the packets into the internal buffers so the latency increases as well.



(a) Packet loss rate

(b) Average latency

Figure 3.13: Fully-meshed traffic performance.

3.4.1.1 Queue occupancy under fully-meshed traffic

Figure 3.13(b) shows for each value of the offered load, the average latency measured for all tester ports. This figure does not contain information about the way this latency varies during the test. This section shows how we used the GETB tester to observe the full range of values of the latency. The latency is tightly coupled with the growth of the queues, as explained next.

Switches are devices that route and forward packets. To handle the cases when multiple packets need to go simultaneously to the same destination, switches use queues (buffers) on the egress, at the output of each port. The occupancy of a queue increases when its input rate exceeds the output rate[38].

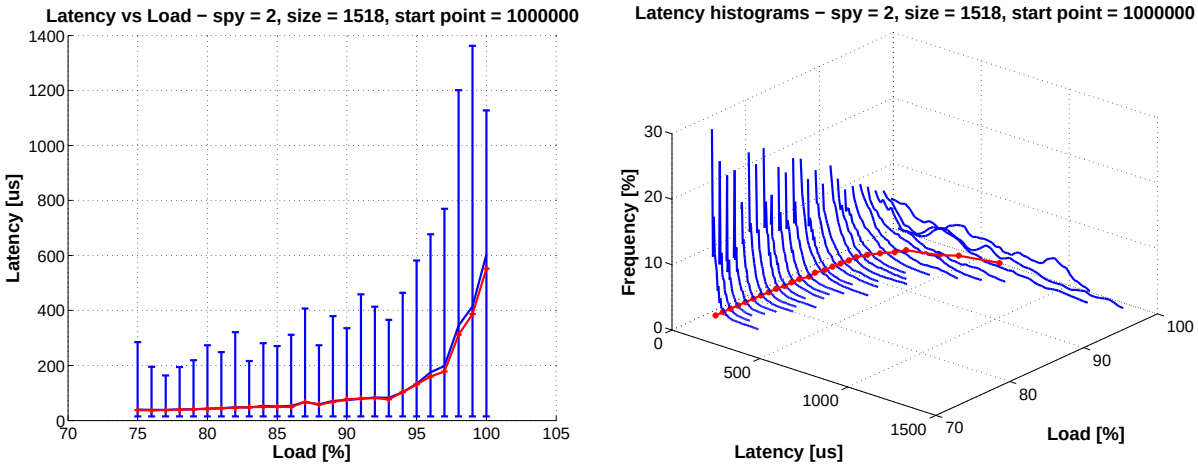
[38]The input rate is a random variable determined by the packets that arrive from other ports. The output rate is fixed and corresponds to the line-rate (Gigabit in our examples).

During a full-mesh test with random choice of the destinations[39], it can happen that a few ports try to send simultaneously to the same destination. This will create congestion and packets will have to wait in the queue of the output port to the target.

The maximum size (depth) of the output queue in a switch is an important factor that affects the way the device handles the "N-to-1" pattern specific to ATLAS. In Section 3.4.2 and in Chapter 4 (Section 4.8.1) we shall present relevant results in this matter. In the following we show how to estimate the instantaneous occupancy of the output buffer.

The latency represents the time needed by a packet to go from the source to the destination. If the packet has to wait in a queue, its latency increases. If there is a single queue where the packet can wait, then we can assume that the latency is directly proportional to the occupancy of the queue.

Using the packet capture mode described in Section 3.3.1.3 we recorded the latency values during a full-mesh run with the largest sized packets. The recording was done on one of the ports participating in the test. For each offered load, we obtained an array of latencies. In Figure 3.14 we show the latency error-bars[40] and the latency histograms corresponding to each load.



(a) Error bars

(b) Histograms

Figure 3.14: Latency variation during a full-mesh test.

In both figures (a and b) we see that the dispersion increases considerably with the load. This is due to the use of a random traffic pattern which makes high and low values of queue occupancies equally probable (for higher loads). At lower loads, the histogram has a bias towards lower values meaning the queues are mostly empty, with only occasional bursts.

The red line represents the average latency. It is the type of curve shown in Figure 3.13(b) for different packet sizes. We observe that for load 100%, while the average stays around 500 μs, there are times when the queue is almost full, increasing the probability of loss.

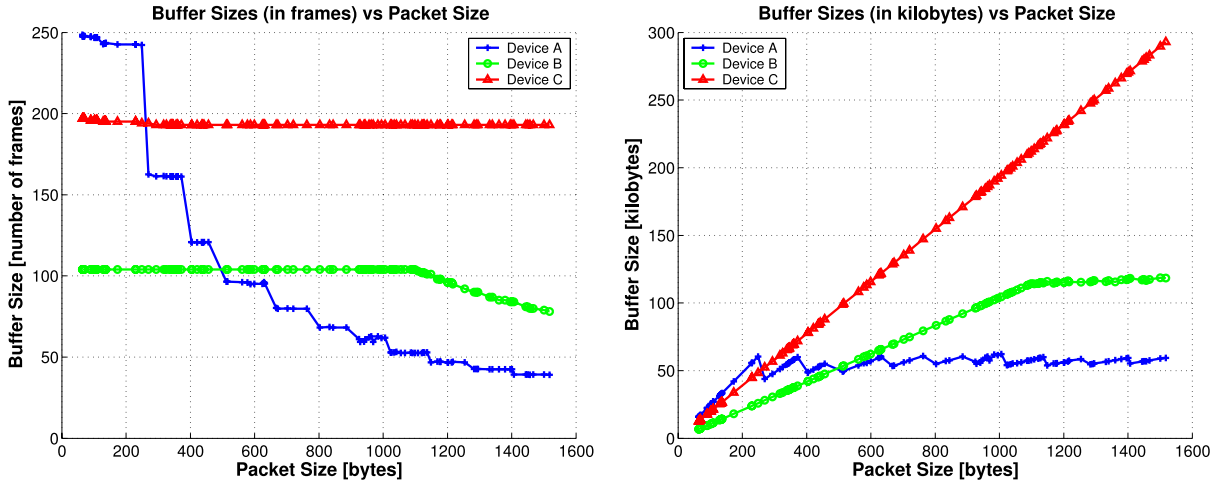
[39]Each node sends packets to all the others, but in a random order.

[40]The length of an error-bar corresponds to the difference between the minimum and maximum latency values.

3.4.2 Buffering capacity and the ATLAS traffic pattern

The ATLAS TDAQ network has several critical points where a number of traffic streams are concentrated to a single destination[41] (a funnel shaped traffic pattern as described in Chapter 2). The performance in such conditions depends on the amount of buffering inside the switch and the way the congestion is resolved.

We developed methods[42] to measure the effective buffering capacity for a port on the switch. Figure 3.15 shows this capacity for different packet sizes and for 3 different devices, expressed in number of frames (Figure 3.15(a)) and in Kbyte (Figure 3.15(b)).



(a) Measured in frames

(b) Measured in kilobytes

Figure 3.15: Buffering capacity.

The shapes of the curves can give us insight to the internal memory allocation policy. For example Device A seems to split and store packets in small (approximately 128 byte) cells, while Device C uses a fixed-size slot for any packet size. Device B uses a linked-list memory management, with a total buffering memory of approximately 120 Kbyte, and a maximum of approximately 105 elements in the list; for small frames (64 to 1100 bytes) the limitation is the number of elements (descriptors) of the list, while for large frames (bigger than 1100 byte) the limitation is the total memory size.

Knowing the buffer size alone is not enough to characterize a device. The way the switch deals with congestion internally is also important. Figure 3.16 shows the loss rate in an ATLAS traffic scenario for 5 different buffering levels; these measurements are taken for a device that

[41] The most critical are the outputs to the SFIs. An SFI sends requests to all the ROSs and then receives back event fragments from 150 sources. All sources are sending almost simultaneously – this may create congestion at the output port to the SFI. The SFIs try to reduce the likelihood of congestion using the token-based traffic shaping mechanism (Section 3.3.1.2, Chapter 4).

[42] See [4], Section 3.8 at page 22.

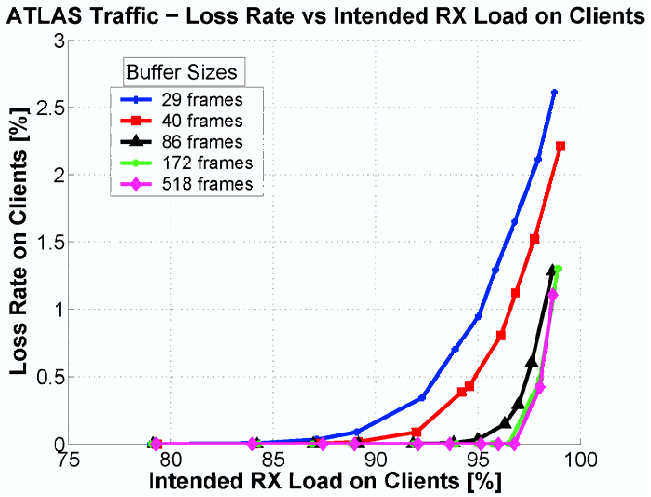


Figure 3.16: Impact of buffer sizes on ATLAS traffic performance.

has user-configurable buffers. The concentration ratio[43] was 6:1 (six times more servers than clients). The x-axis represents the intended load to be obtained on the congested ports, and the loss rate is shown on the y-axis. We observe that for small numbers of available buffers, packet loss starts to appear at 85% load and this onset increases to 95% when more buffers are allocated. Although the loss rate is not large (2-3%), it carries a substantial performance penalty in the request-response dialog in ATLAS (due to the time lost in timeouts followed by re-asks for data).

3.5 Other applications

In addition to Network Tester, two other complete applications have been developed on the GETB platform. This proves the versatility of the system. The tester and these applications share a part of the FPGA firmware of the entire control software running on the PC.

3.5.1 The ATLAS ROB emulator

The interface between the ATLAS detectors and the TDAQ is the Read Out Buffer (ROB), an intelligent buffer management card that receives event fragments from detectors on three optical links, storing the fragments in local memory. The ROB responds to requests for these fragments from the L2PUs and SFIs. There will be over 500 ROBs in ATLAS.

In order to validate the software on the L2PUs and on the SFIs, we implemented a ROB Emulator[44] using the GETB. This application runs on the FPGA and delivers data on demand

[43]The concentration ratio represents the number of servers over the number of clients, keeping the terminology from Section 3.3.1.2.

[44]The ROB emulator was elaborated by my colleague, Dr. Micheal LeVine. The author provided assistance on

to the TDAQ applications. It complies with the ATLAS event format, which allows a seamless integration with the software programs. As the ROB Emulator can run at Gigabit line-speed, it allows the ATLAS software developers to test their applications and check if they can attain and sustain line-speed performance.

3.5.2 Network emulator

A *Network Emulator* is a system that makes it possible to study in a laboratory setup real applications under variable network conditions. Using a network emulator, the user has access to a "network in a box" that allows to control the end-to-end quality degradation. By controlling the amount and type of degradation introduced by the emulator, one can study the application behavior in a wide range of network conditions (see Figure 3.17).

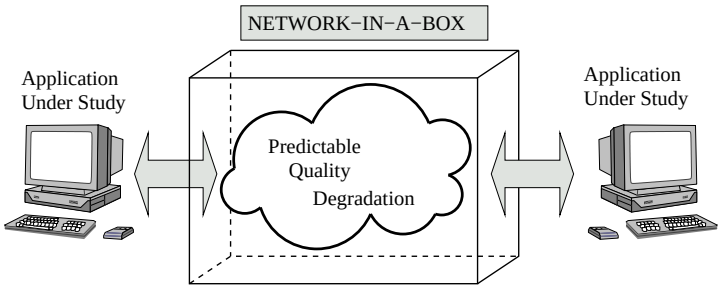


Figure 3.17: The network emulation concept.

A hardware network emulator has been implemented using the GETB platform[45]. The GETB card is used in pass-through mode (traffic flows between the two ports of the card). The network emulator uses a system of queues to provide realistic network conditions where the packet loss and the delay are correlated. The user can define criteria to classify and differentiate between multiple traffic streams and then apply different degradation models to each stream. Using this feature one can emulate the service differentiation used in some routers. To emulate the degradation introduced by other traffic flows in a network, background traffic generation is supported. Inside the emulator the virtual background traffic will be mixed with the traffic flow of the main application. In order to emulate the effects of overload and those of transmission over low-speed links, a mechanism of rate limitation is also available.

3.6 Summary

In this chapter we presented the GETB, a platform that provides a flexible environment for the design and development of Gigabit Ethernet applications. A network tester capable of generating

the use of the GETB core firmware and of the control infrastructure.

[45]The Network Emulator has been developed by my colleagues Dr. Mihai Ivanovici, Dr. Razvan Beuran and Prof. Dr. Neil Davies. For an in-depth presentation of the Emulator, please refer to the PhD thesis of Mr. Ivanovici [40].

traffic similar to the one produced by the ATLAS TDAQ software has been designed and implemented based on the GETB. The tool proved to be useful and efficient in evaluating switches for the TDAQ network. In addition, the versatility of the GETB has been demonstrated: two other network-related applications have been implemented on top of this platform. The next chapter shows another application of the GETB tester – using it as a tool for studying traffic patterns.

Chapter 4

An analysis of the ATLAS traffic pattern

4.1 Introduction

In this chapter we analyze the traffic pattern used by the data acquisition (DAQ) applications in ATLAS. In order to improve the fault-tolerance and to maximize the efficiency in a given set of network conditions, ATLAS has opted for a request-response protocol. In the next sections we discuss the characteristics of this type of traffic.

We begin in Section 4.2 with a description of the mechanism that is used to regulate the data rates. Then we continue in Section 4.3 by presenting the tools we used for this study – both in terms of hardware and software. Sample measurements and observations are shown at the end of this part.

Starting from the rules that govern the requester (the client), we derive in Section 4.4 an analytic formula which allows us to compute the average input rate it perceives. Then we present a comparison between our theory and measurements. Section 4.5 is based on results obtained on a pure hardware test-bed. In this case, the behavior of the system is fully deterministic and the predictions match very well the theory. Queueing is also studied from the perspective of a hardware implementation (Section 4.6). The second set of observations comes from a software test-bed; deviations from the theory become visible in this case (Section 4.7). Finally, in Section 4.8 we present practical applications of the tools we've developed and we draw the conclusions.

4.2 Traffic shaping

Request-response protocols tend to use efficiently the network bandwidth and the computing resources. With such protocols, data is transferred only when it is needed. The final user is free to decide when and if the data gets transferred. An example is the Hypertext Transfer Protocol (HTTP) used for web pages. In HTTP, the constituents of a website (images, animations, etc.)

are retrieved sequentially when they have to be displayed in the browser.

As we have seen in Chapter 2, in the ATLAS DAQ, data flows from the detector down to mass storage through several layers of filtering. At any given moment some part of the system is busy analyzing or collecting a physics event. The filtering processes work in a "soft" real-time mode. They may spend more time on some events than on others (within certain limits). Therefore, it is important not to saturate the filtering layers with too much data. In addition, the network responsible for moving the data between the applications has a finite capacity which is much lower than what the detector can produce[1]. Some means to regulate the traffic are necessary.

In the early design phases there have been two proposals for ATLAS. One was the "push" scenario, in which events were sent to a filtering node, without any feedback from the node itself or from the network. The amount of traffic was controlled by enforcing *Quality of Service* rules throughout the system (at the network level or at the operating system level). The load balancing was done via the DataFlow Manager (DFM) which was selecting the nodes (Sub-Farm Interfaces) to act as sinks for the detector buffers (Read Out Buffers/Systems). In the "push" scenario, self adjustments of the rate in case of congestion were not possible.

The alternative was the "pull" scenario – this was chosen for the final ATLAS. Here, data from the detector is delivered only on request from the filtering nodes. The rate is determined by the requesters. In the following we denote as *clients* the nodes requesting data and as *servers* the ones providing it. On one hand, the clients should send requests fast enough in order to use the network at full capacity, on the other hand they should be careful not to request more data than they can process (or more than the network can transfer).

Event fragments are spread across many servers (the ROSs) and the requests need to go to all of them. These messages consist of small packets and they arrive quite fast at the servers. The replies with event data are much larger[2]. For any client, the replies come from different locations and they all head to the same destination – network congestion can appear in this case. We call this a "funnel-shaped" traffic pattern (Figure 4.1), in which many sources are simultaneously sending to one destination.

The client can very easily over-drive the system, by asking for too much data (it sends several requests and then gets overwhelmed by the quantity of data it gets back). A mechanism to control the flow has to be devised – such a mechanism is called *traffic shaping*. In ATLAS, the clients control the rate by establishing an upper limit on the number of outstanding (unserved) requests. The upper limit depends on the capacity of the network and of the node itself.

When the client does not get all the replies it has requested, it has to re-ask for the missing ones – this happens after the period of a timeout (a reply might be simply late). Any packet loss causes timeouts; they induce a serious drop in performance. The client has to be configured to avoid losses, but still let the system run at a reasonable rate.

[1] ATLAS delivers event data at a rate of 64 Tbyte/s. The TDAQ network can sustain only several hundred Gbit/s.

[2] The typical request message fits into one 64 byte packet. An event fragment has usually 12 Kbyte, so it spans over 7 or 8 packets of 1518 bytes.

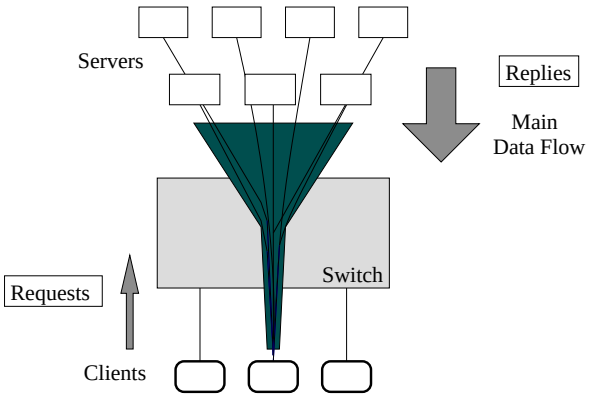


Figure 4.1: The funnel-shaped request-response pattern.

4.2.1 Tokens and watermarks (L_W, H_W)

As said above, we can place an upper limit on the traffic level by limiting the number of unserved requests. The client could simply send a number of N requests, wait for all of them to be served and then start again the cycle. If this is done, the network will be mostly idle in the direction of the main data flow (servers to clients). As an improvement, the client could start sending new requests *before* getting all the replies. If it asks for data in advance then it will have to wait less so the average rate should increase.

Based on these observations, we specify now the set of rules that govern the traffic shaping mechanism employed in the ATLAS DAQ. It is based on *tokens*, where a token corresponds to an outstanding request sent by a client. Any client has to obey to these rules.

- The client starts with zero tokens. It gains a token when it sends a request and loses a token when it receives a reply. The client keeps track of an internal token counter (T_C) for the current number of tokens. The number of tokens is bounded by two watermarks.
- The *High Watermark* (H_W) represents the maximum number of tokens that the client is allowed to have (i.e. the maximum number of unserved requests).
- The client sends requests until it reaches the High Watermark and then it listens for replies, decreasing the token counter by one, for each reply.
- When the number of tokens reaches a *Low Watermark* (L_W), the client can resume sending requests, as fast as possible, until the High Watermark is reached again. Then the cycle repeats, with the client waiting for enough replies to be granted the right to send again.

Figure 4.2 shows how the token counter evolves in time. First it reaches H_W (line (1) from top-left to top-right), then the transmission of requests is disabled (segment (2)), then replies start to arrive (segment (3)) and T_C reaches L_W; finally transmission resumes (segment (4)). We observe that the token counter oscillates between L_W and H_W (the "running" loop in the figure). When the client stops, the token counter comes back to zero. This traffic shaping mechanism is efficient if the parameters (L_W, H_W) are tuned to the specific network conditions.

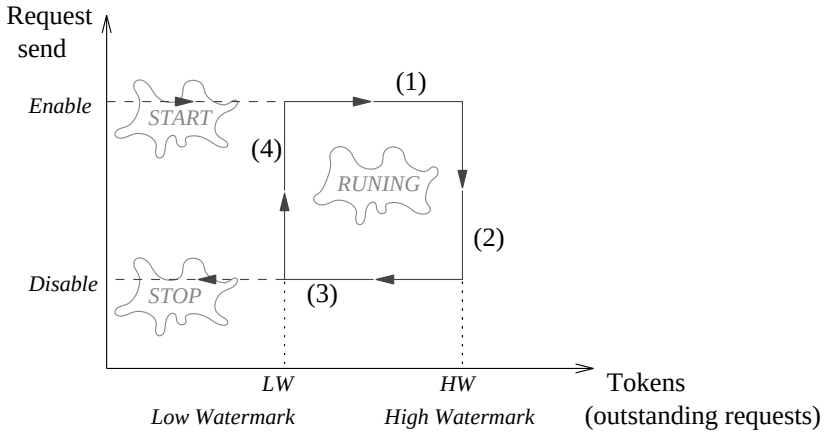


Figure 4.2: The traffic shaping cycle.

4.2.2 Usage in ATLAS

The central part of the ATLAS DAQ system (the Level 2 and the Event Building) uses token-based traffic shaping. This part of the TDAQ has the highest demands in terms of network bandwidth and efficiency (small losses and delays).

We explained in Section 2.1 that the data from the detector is buffered into the Read Out Systems (ROSs). Then a fraction of this data is analyzed by the Level 2 Processing Units (L2PUs). For the accepted events, the Sub-Farm Interfaces (SFIs) collect the event fragments from all the ROSs. With the terminology from the previous section, in ATLAS, the servers are the ROS computers, while the clients are the L2PUs and the SFIs.

The L2PU runs a *data analysis* application – its task is to get a few event fragments from the ROSs and quickly decide whether or not to accept an event. In order to make its decision, the L2PU needs all the fragments it requested – it will not ask for more data unless it has finished with the current set. From the point of view of the network, this behavior can be described by setting the High Watermark equal to the number of fragments needed for analysis and the Low Watermark to zero. In order to optimize the throughput, the L2PU computer can run several L2PU applications simultaneously – the effect from the network point of view is an increase of both watermarks (so L_W becomes non-zero).

The SFI is different from the L2PU – it runs a *data collection* application. It aims to get all the event fragments from all ROSs as fast as possible. But the SFI has to take into account the capacity of the network so as to avoid packet loss (and timeouts followed by re-asks). The SFI uses an "aggressive" configuration for the tokens – it tries to keep a constant number of outstanding requests in the network. This number is called the *Traffic Shaping Parameter* (TS). As soon as it gets a reply, the SFI immediately issues another request. This behavior can be described by setting $H_W = TS$ and $L_W = H_W - 1$. Inside the SFI, the token counter is shared (accessed) by the transmission and reception tasks (threads) – a semaphore protects this counter from data corruption. When the replies are small, the token counter has to be updated very often. Frequent accesses to a semaphore-protected variable can slow down the application.

To overcome this problem, the SFI can be configured to update the token counter only if the update will change the counter by at least N units[3]. This translates into the following setting for the watermarks: $H_W = TS$ and $L_W = H_W - N$.

In both cases, for the "ROS to L2PU" and "ROS to SFI" communication, a client requests data from several servers. This setup is likely to cause network congestion because of the funnel pattern created in the direction of the clients.

What we aim to find out in this chapter is how the traffic shaping parameters influence the network traffic. What can we expect in terms of network bandwidth when we set certain watermark values? What are the "best" values in a given network configuration? What are the dependencies of these values? Can we know in advance if the TDAQ system will work in a given network setup and with a certain software configuration? Answers to these questions will be given along the pages of this chapter.

4.3 Testing equipment and methodology

A custom test-bed has been created to study the token-based traffic shaping. A test-bed is a laboratory environment where the user is able to adjust the "inputs" (watermarks, number of clients and servers, network topology, etc) and to measure precisely the "outputs" (throughput, delays, losses, etc.) In our case we needed to generate network traffic according to the rules from Section 4.2.1. Two different traffic generators have been developed for this purpose: one based on the GETB hardware platform (described in Chapter 3) and one based on software programs. The "network" we employed for our studies consisted of a single Gigabit Ethernet switch. The next two sections bring more details.

4.3.1 GETB, traffic-shaping in hardware

Our first measurements were done using the GETB traffic generator (Section 3.3). The GETB implements in hardware (FPGA) the traffic shaping mechanism – this makes it very accurate and fully deterministic. To configure a tester port as a client, one needs to specify the watermarks, the size of a request and of a reply[4] (usually 64 and 1518 bytes) and the list of servers where to ask for data. The ports acting as servers do not require any special configuration – they just answer to the incoming requests. Once the clients are started, the GETB can deliver statistics on demand (throughput, one-way delays, amount of packet loss).

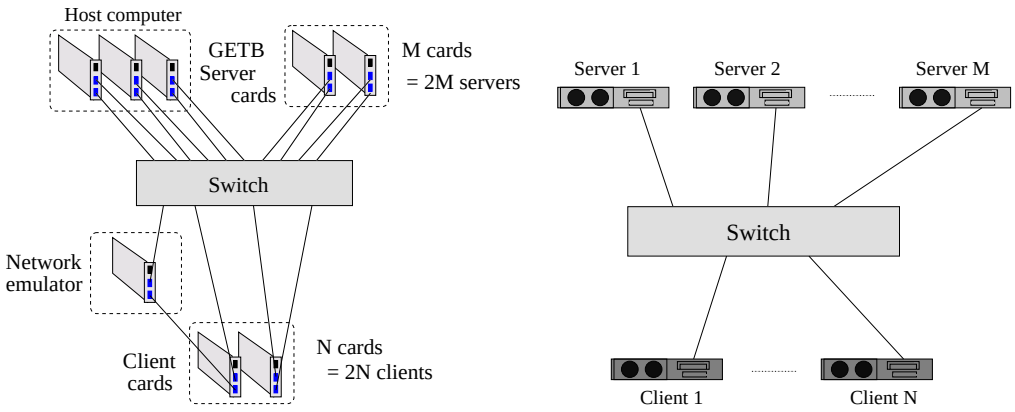
For some of the measurements, we activated also the packet capture mode (Section 3.3.1.4). This feature permits not only to see the arrival time and the latency for each packet, but also to have the information about the number of tokens in the client and the internal queue occupancy in the server, on a packet by packet basis (the capture mode proved to be very useful when we

[3]This parameter is called *Semaphore Concentration* in the SFI.

[4]For additional flexibility, we specify the size of a reply at the client. In this way, the client can ask for replies of different sizes during a test. In the real ATLAS the size of a reply is fixed and corresponds to the size of an event fragment.

examined queueing).

In order to see the effect of different network conditions, we employed also the GETB network emulator (Section 3.5.2). We used it in the "fixed delay" mode where it behaves like a long line (adds a large delay to all packets). Figure 4.3(a) shows a block diagram of the GETB test-bed: with $M + N$ GETB cards we can instantiate $2 \cdot (M + N)$ servers and clients.



(a) Based on GETB cards

(b) Based on *mpNetPerf* applications

Figure 4.3: Test-beds used to study request-response traffic.

4.3.2 mpNetPerf, a software implementation

With a hardware platform like the GETB, the rates, delays and losses are determined only by the network. Most of the ATLAS DAQ, however, depends on software programs. For the completeness of the study, we've developed tools to investigate a software implementation of the request-response traffic. These tools try to emulate the behavior of real DAQ components.

In a complex infrastructure like that of the ATLAS experiment, which consists of a large number of projects and developers, minimizing code redundancy is essential. That's why there exist many software libraries shared among different applications; they provide various kinds of common services: message and error reporting, operational monitoring, run control, etc. Network communication is one such service and ATLAS has a common software layer for this purpose. The *Message Passing Library* is responsible for the exchange of messages over the network – the SFI, L2PU and ROS applications are based on it. The physics events from the detector are encapsulated into messages and transferred using request-response protocols built on top of the *Message Passing*.

For the purpose of our study, we developed a set of programs which link to the Message Passing library and generate request-response traffic. These tools, gathered under the label *mpNetPerf* [5], can stress the network more thoroughly than their "real" counterparts (because

[5]Message Passing Network Performance Tools.

they do not perform any data processing). They are meant to devise an upper bound for the level of performance attainable in the TDAQ system[6].

mpNetPerf uses the ATLAS Message Passing to send and receive requests and responses. The communication library takes care of the operating system calls that do the actual network transfers. Two network protocols are transparently supported: UDP and TCP. As in the case of the GETB tester, an *mpNetPerf* instance can be either a client or a server. The client issues requests to servers and adjusts the rate using a token-based system. *mpNetPerf* does not use a global clocking scheme, so it cannot measure one-way delays. Instead, the client records the *Response Time* or *Round-Trip Time* (RTT) for each request-response transaction.

The implementation of *mpNetPerf* does not make use of features specific to a real-time process, therefore its performance can degrade due to factors like CPU contention, inter-thread communication[7], lack of network bandwidth, etc. The software test-bed is depicted in Figure 4.3(b), the difference with respect to the GETB being that the clients and servers are now standard PCs: now with $M + N$ computers we can have only $M + N$ servers and clients.

4.3.3 Dealing with packet loss

Data loss can occur at any time in an Ethernet network – higher level protocols are supposed to insure reliable transmission. Within a request-response protocol, we can have losses in both directions, as described below:

- *Missing requests.* Losing requests is unlikely as they travel at small rates on the network. In ATLAS, the servers[8] cannot do much if they detect a lost request. The testing tools we've developed can recover in such a situation. Each request contains a sequence number which makes it possible for a server to detect losses (gaps in the sequence). After learning how many messages have been lost from a particular client, the server can "fill-in" the replies which correspond to the requests that were never received[9]. In the real ATLAS, a lost request translates into a missing reply and then into a timeout at the client (and the request being regenerated).

- *Missing replies.* This usually happens because of network congestion – the watermarks are not configured properly and the network cannot buffer all the incoming reply messages. The ATLAS clients[10] use timeouts to deal with this situation. In GETB and *mpNetPerf*, the clients detect the losses based on sequence numbers and adjust the token counters to

[6]The "system" is composed of the network *and* the computing cluster where instances of *mpNetPerf* are running. This is different from the GETB case, which measures only the performance of the network itself.

[7]*mpNetPerf* has two threads for the reception (RX) and transmission (TX) of messages. The threads in the client share a common resource, the token counter – this is protected by a semaphore. The performance of the client is partially limited by the frequent accesses to this counter, which requires blocking of one of the threads. In the server, the TX and RX threads communicate through a queue (for requests). This also requires a locking mechanism.

[8]The Read Out System (ROS) computers. They contain the ROB cards which buffer event data from the detector.

[9]This is not possible in a real system where the replies contain meaningful data, specific to a request.

[10]The Sub-Farm Interfaces (SFIs) or the Level 2 processors (L2PUs).

compensate for the missing replies (they pretend the replies were received) – there are no timeouts, nor re-asks.

Note that the "error-recovery" implemented in our test tools makes the traffic shaping mechanism ineffective. Normally, this mechanism is meant to cope with network congestion. When buffers in the network are full, the delays increase and the clients gradually reduce the request sending rate (because they have to wait more for replies), so the congestion will disappear. By recovering from losses, our tools practically cut the feedback from the network. This design decision was made just to be able to stress the network at the maximum (even beyond the break point). Another reason was to reduce the complexity of the GETB / FPGA implementation[11].

Losses can appear not only in the network, but also at the end-node – in the operating system (OS) or at the application level. This can occur at reception, if the node is too busy, but also on transmission, if the node tries to send too fast. With the current technology, personal computers can easily transmit at Gigabit line-speed. When a connection-less oriented protocol is used, such as UDP, many operating systems do not put any restriction on the sending rate of the software application. With UDP, the application opens a socket and then calls the *send()* system call. When the output line is fully utilized, the system call should fail, so the application is notified and can slow down. Our experiments have shown that this does not happen in Linux for example – the system call succeeds and the OS simply discards the messages[12]. There exist operating systems that provide a feature called *intra-stack flow control*[13]. This is supposed to propagate the congestion from the lower layers (network interface in this case) up to the operating system stack and then to the application. As ATLAS will be based on Linux, the only way to make sure the servers are not trying to send faster than line-speed is to adjust the token watermarks accordingly.

4.3.4 Preliminary results

We present now two plots obtained on our test-bed (the one based on the GETB hardware-based traffic generator). As the traffic shaping rules are supposed to determine the data rate perceived by a client, this will be the parameter we will focus on. We shall denote it as *RX Speed* [14]. The curves from Figure 4.4 show the dependency of the *RX Speed* on one watermark, while keeping the other one constant. In these and in all subsequent plots, the *RX Speed* is measured in percents of line-speed[15]. For Figure 4.4(a) we increased the High Watermark while the Low Watermark was set to zero. We notice a slow increase of the rate perceived by the client and also that the curve bends before reaching 100% of the line capacity. For the second figure we

[11]Each request / reply message has enough information to let the server / client recover from a lost message. In this way we don't have to maintain in memory a table with the "state" (progress) of all request-reply transactions and to re-ask for data when there are losses.

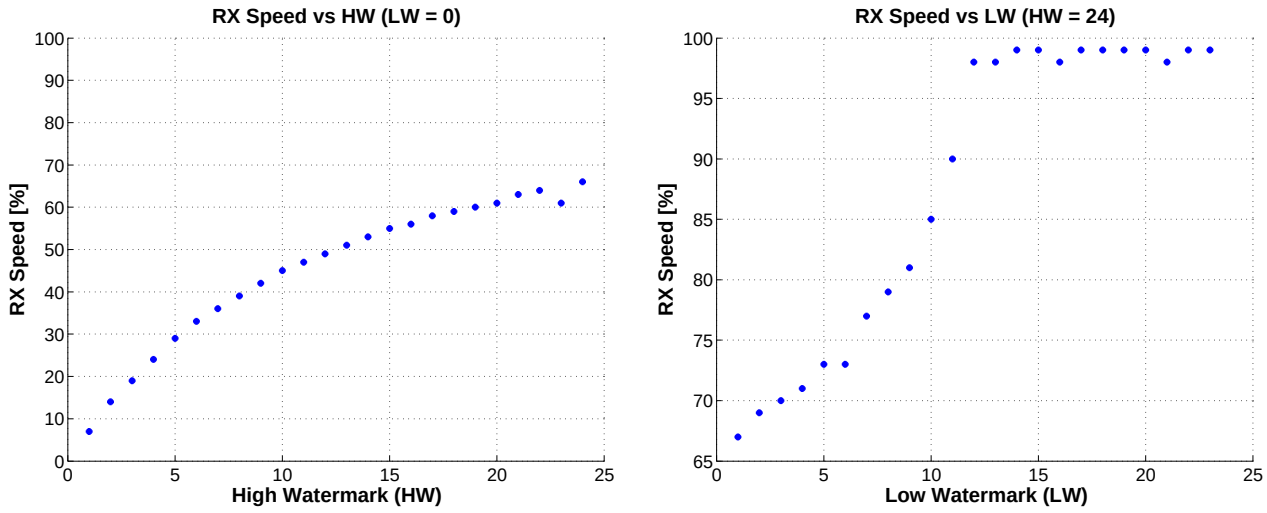
[12]This behavior has been observed on Linux with kernel version 2.4.21. This particular version does not seem to keep any statistics on the number of such "internal drops".

[13]Sun's Solaris seems to support intra-stack flow control.

[14]The *RX Speed* can be expressed in bytes per second, bits per second or simply as a percentage of the maximum line-speed.

[15]Unless otherwise noted, all our results were obtained using Gigabit Ethernet links.

had $H_W = 24$ and we changed L_W. Line-speed is attained faster this time. In the next section we explain these plots and we derive an analytical expression for the *RX Speed*.



(a) LW=0

(b) HW=24

Figure 4.4: Preliminary measurements for RX Speed.

4.4 The expected input rate

The expected input rate at the client is a function that depends clearly on the two traffic shaping watermarks, L_W and H_W. It is also influenced by the size of the replies (S_{rpl}) – bigger replies adding more load to the network. Measurements revealed a correlation between the *RX Speed* and the network conditions, more exactly on the delays introduced. Our goal, in this section, is to find the function:

$$Rx = f\left(L_W, H_W; S_{rpl}, \ldots \text{network} \ldots\right) \tag{4.1}$$

Note that expression (4.1) does not contain as an argument the number of nodes in the system (clients or servers). The reason for this we'll be explained later.

4.4.1 Initial approximations

We start from the definition of "speed" or "rate" as the quantity of data (ΔQ bytes) transferred in a given time interval Δt:

$$\text{Speed} = \frac{\Delta Q}{\Delta t} \tag{4.2}$$

We tackle first the cases with one variable watermark; this will lead us to a more general expression. We'll make the following assumptions:

- The number of clients is always smaller or equal to the number of servers.
- There is no other traffic in the network, apart from request-reply transactions.
- The servers and the clients are all connected to the same network switch.
- The requests from a client are uniformly distributed[16] to all servers.
- The size of a request is negligible compared to the size of a reply.
- There is no processing delay at the server – it can answer as soon as it gets a request.

In addition to these, in the analysis that follows we consider that we have one client connected via an Ethernet switch to only one server. This does not limit the applicability of the results, as we shall see later.

4.4.1.1 $L_W = 0, H_W = N$

In order to apply (4.2) to the request-response traffic, we need a better understanding of the way data travels over the wire. When $L_W = 0$ and $H_W = N$, the client begins by sending N requests, as fast as possible. The first request arrives at the server after D_{req} seconds, the delay introduced by the network. If the network consists of a single switch, then D_{req} will be the switching delay of a packet of size S_{req} bytes.

Assuming there is no processing delay[17], then the server immediately begins answering with a reply of S_{rpl} bytes. After D_{rpl} seconds, the propagation delay of the reply via the network, the client gets the first answer.

We call $D_{req} + D_{rpl}$, the *Base Round-Trip Time (T_0)*; it is the time until the client gets the first reply. The client then continues sending requests until the token counter reaches the high watermark, $H_W = N$. This happens in $N \cdot T_{req}$ seconds, where T_{req} is the time to send one request (transfer it from the host to the network[18]). This time is usually much smaller than T_0; in addition, the server needs more time to send a reply than to receive a request ($T_{rpl} > T_{req}$). Effectively, the client has enough time to send *all* requests before even receiving the first reply[19]. The server cannot handle requests as fast as they arrive (its service time has T_{rpl} as the lower bound) so it places them into an *internal queue*. The server sees requests 2 to N ready, waiting to be served in the queue – the corresponding replies are transmitted in a burst, as fast as possible. Each one needs T_{rpl} seconds to be sent, then another D_{rpl} seconds to travel over the network

[16]The distribution can be random (uniform) or deterministic (round-robin).

[17]This is the case for a hardware implementation like in the GETB tester. In software, in *mpNetPerf* for example, the operating system will introduce additional delays.

[18]The transfer time depends on the speed of the link. For example, a packet of 1518 bytes is transferred in 12.3 μs on a Gigabit Ethernet line. The transfer time is the same for reception and for transmission.

[19]Common values are $T_{req} = 0.67\mu s$, $T_{rpl} = 12.3\mu s$, $D_{req} = 7\mu s$, $D_{rpl} = 20\mu s$, $T_0 > 25\mu s$ and $H_W = N = 10$.

and then T_{rpl} seconds to be fully received by the client. The client sees replies arriving at T_{rpl} intervals; the other delays are hidden because the network works as a pipeline.

Therefore, the client gets the first reply after T_0 seconds and the subsequent ones at line-speed (as they were sent). The entire set of N replies is received in $N \cdot T_{rpl}$ seconds. After getting all replies, the client hits the lower limit $L_W = 0$ and starts another cycle, sending another bunch of N requests. This cycle will repeat forever so the rate can be computed for one period.

The time to complete a cycle from the first request sent to the last reply received is $T_0 + N \cdot T_{rpl}$ seconds. The amount of data transferred during this period is $N \cdot S_{rpl}$ bytes. Therefore the input rate is[20]:

$$Rx\left(L_W = 0, H_W = N; S_{rpl}, T_{rpl}, T_0\right) = \frac{N \cdot S_{rpl}}{T_0 + N \cdot T_{rpl}} \tag{4.3}$$

During one cycle, there is a period of silence which lasts T_0 seconds followed by a burst of replies coming at line-speed. In Figure 4.5(top) we present a packet trace recorded at a client configured with $L_W = 0, H_W = 8$. On the Y-axis we have the token counter, T_C at the client; for each reply (red point) the T_C is decremented. The figure shows a few bursts and the gaps of length T_0 separating them. We shall see in the following section that the inter-burst gap can be shorter, hence bursts closer, increasing the average rate. This is why we shall refer sometimes to T_0 as the *Maximal Inter-burst Gap*.

At this point we can explain the curve from Figure 4.4(a). We understand why the curve saturates as the High Watermark increases (N in our formula) and never reaches line-speed due to the constant factor (T_0) at the denominator. With our notations, we can "define" now the line-speed as:

$$\text{Network line-speed} = \frac{S_{rpl}}{T_{rpl}} \tag{4.4}$$

4.4.1.2 $L_W = 1, H_W = N$

The formula (4.3) applies only for $L_W = 0$. When $L_W > 0$ the client resumes sending requests while it is still getting replies from the server. Suppose $L_W = 1$ and $H_W = N$. Then, when $N-1$ replies have been received, and there is one more to get (the token counter being $T_C = L_W = 1$), the client *refills*[21] the server with $N - 1$ new requests. Immediately T_C jumps back to N. The server gets now only $N - 1$ requests per bunch, not N as in the case with $L_W = 0$.

After sending the $N - 1$ requests, the client receives the last reply from the first batch (in T_{rpl} seconds) and then starts waiting for the replies of the second batch. It cannot send new requests because it has $T_C = N - 1$ and is allowed to send only when $T_C \leq 1$.

[20] In this formula, T_0 is the network dependent parameter. The size S_{req} does not appear because our assumption was that requests are much smaller than the replies, hence their network transfer time is negligible.

[21] We shall call *refilling* the process of sending a set of requests until the maximum upper limit is reached (the high watermark).

When $L_W = 0$ (Section 4.4.1.1) the time between two bursts of replies was T_0. Now, as the client has started *refilling* earlier, this time decreases accordingly. The client has to wait only for $T_0 - T_{rpl}$ seconds, i.e. the initial time (T_0) minus what was saved by sending a request a little bit earlier. After this interval, another burst of replies arrives – it is shorter, with only $N - 1$ replies. Towards the end of the second burst, the client issues another set of $N - 1$ requests and the cycle restarts. The client still keeps N outstanding requests, but now makes sure that one of them ($L_W = 1$) is always in transit.

Figure 4.5(bottom) is the packet trace at the client, showing the moments when replies are being received; we remark the decrease of the inter-burst gap and of the burst length.

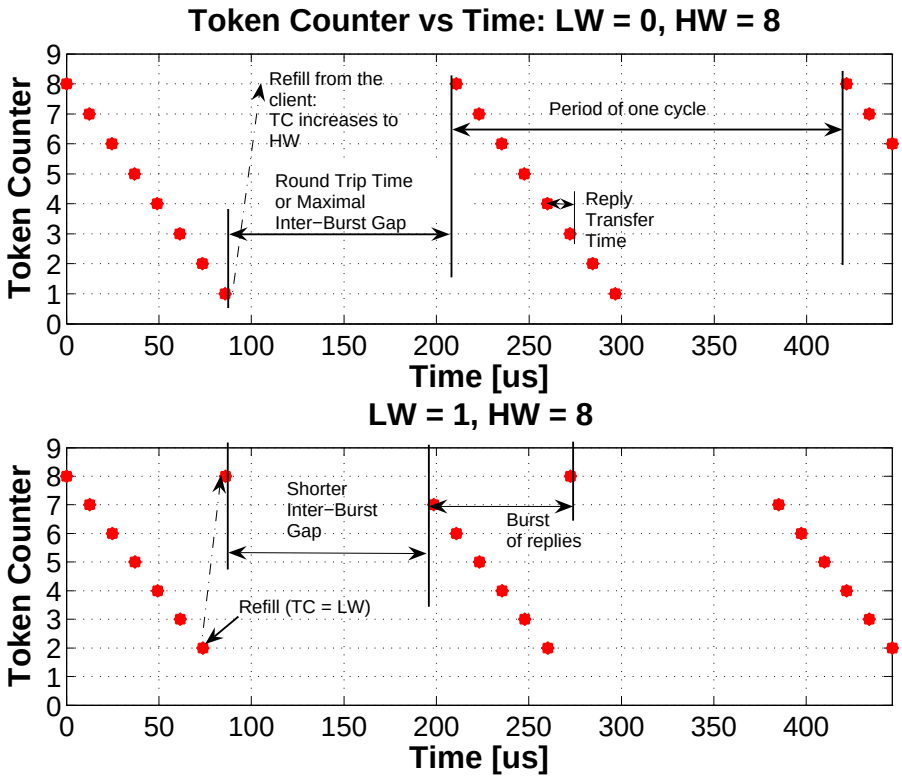


Figure 4.5: Token counter vs Time: L_W =0 and 1, H_W =8.

To find the rate, we can use the same line of reasoning as for $L_W = 0$. There are two differences: a shorter inter-burst gap ($T_0 - T_{rpl}$) and a shorter burst length ($N - 1$ instead of N). The amount of data transferred during each cycle is reduced by the size of one reply[22]. We can write the following formula for the expected *RX Speed*:

$$Rx\left(L_W = 1, H_W = N; S_{rpl}, T_{rpl}, T_0\right) = \frac{(N-1) \cdot S_{rpl}}{(T_0 - T_{rpl}) + (N-1) \cdot T_{rpl}} \tag{4.5}$$

[22]Except for the first and the last cycles, when the client starts and stops. These will always have N replies.

4.4.1.3 $L_W = M, H_W = N$

If $L_W = M < N$, we can expect that the behavior described in the previous section still applies. The client sends requests earlier, the gap between consecutive bursts decreases with $M \cdot T_{rpl}$ and the length of a burst shrinks also with M. Therefore we can assume the following dependency for *RX Speed*:

$$Rx\left(L_W = M, H_W = N; S_{rpl}, T_{rpl}, T_0\right) = \frac{\overbrace{(N-M)}^{\text{Burst Length}} \cdot S_{rpl}}{\underbrace{(T_0 - M \cdot T_{rpl})}_{\text{Inter-Burst Gap}} + \underbrace{(N-M)}_{\text{Burst Length}} \cdot T_{rpl}} \tag{4.6}$$

4.4.2 Corrections

Measurements have shown that (4.6) is wrong. It contains two mistakes that will be corrected in the following sections.

4.4.2.1 The Inter-Burst Gap

The first one is easy to spot: the time between two bursts cannot be negative. If $L_W \cdot T_{rpl} > T_0$ then the first term at the denominator becomes less than zero. The interpretation is that the bursts are coming back to back, without any spacing between them. The first correction will be to apply the *ramp function*[23] to this term:

$$\text{Inter-Burst Gap} = R(T_0 - L_W \cdot T_{rpl}) \tag{4.7}$$

The low watermark can be thought of as the number of requests or replies "in transit" between the client and the server. For a fixed T_0 we can have at most $\frac{T_0}{T_{rpl}}$ replies/requests traveling across the network. We shall call this the *Network Transfer Capacity*[24] (N_C):

$$N_C = \frac{T_0}{T_{rpl}} \tag{4.8}$$

If $L_W > N_C$, the network (transfer) capacity has been exceeded and requests or replies need to be buffered because they cannot all be in transit. These topics are discussed in Section 4.6.

4.4.2.2 The Burst Length

The second mistake in (4.6) requires a more careful analysis. The problem is that the burst length is not always equal to $H_W - L_W = N - M$. We provide an explanation using the two

[23] The ramp function is defined as $R(x \geq 0) = x$ and $R(x < 0) = 0$.

[24] We shall sometimes omit the word "transfer" when referring to the Network Transfer Capacity.

traces shown in Figure 4.6.

Figure 4.6(top) contains the evolution of the token counter at a client configured with $L_W = 3$ and $H_W = 8$ (client starts sending requests when there are 3 pending replies). After the initial burst of 8 requests (at startup) the client starts receiving replies and T_C decreases (the beginning of the X axis). When $T_C = 3$, the client sends $H_W - L_W = 5$ more requests, at the moment marked with *Q1* (1^{st} refill). Then, after receiving the 3 remaining replies from the first set of 8 requests, $T_C = 5$ and the client starts waiting for the other 5 replies (answers to *Q1*) to come back. They arrive after $T_0 - 3 \cdot T_{rpl}$ seconds. After the first 2, $T_C = L_W = 3$ and the client refills the server with another set of 5 requests (the 2^{nd} refill, *Q2*). Then the client gets the last 3 replies from the first set and waits an entire burst gap for the replies corresponding to the second refill.

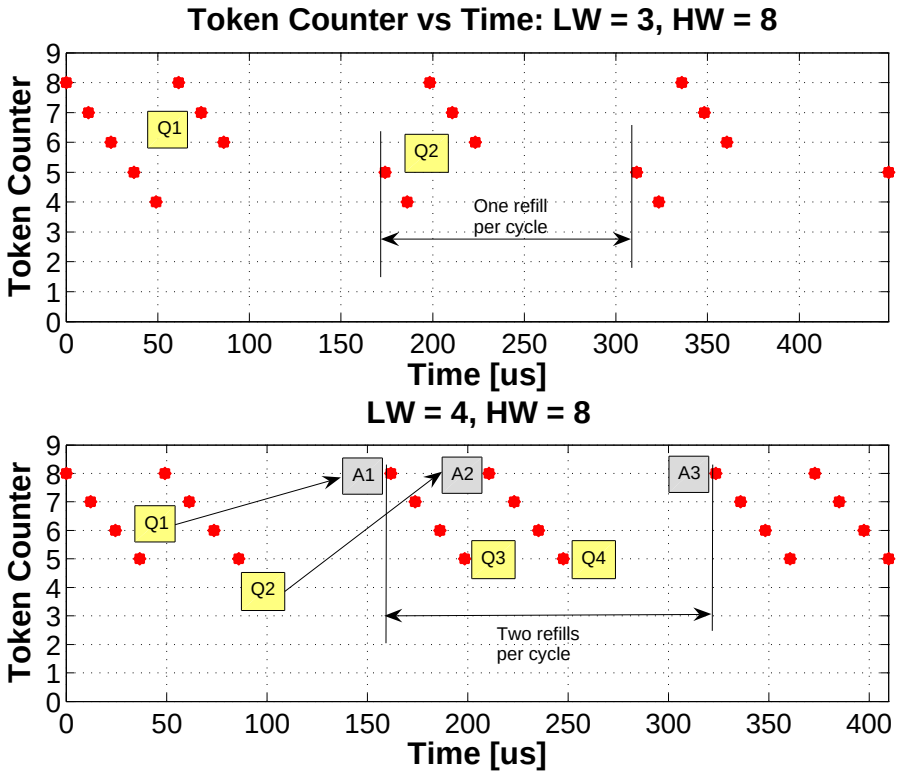


Figure 4.6: Token counter vs Time: L_W =3 and 4, H_W =8.

In Figure 4.6(top) there is always a *single* refill per cycle. We should also keep in mind that the time between a refill operation and its effect (the arrival of the replies) is constant and is equal to the inter-burst gap ($T_0 - 3 \cdot T_{rpl}$ in this case).

In Figure 4.6(bottom) we show another trace for a client with $L_W = 4$ and $H_W = 8$. Again T_C starts at 8 and decreases. When $T_C = 4$ (marked with *Q1* in the figure) the client does a complete refill and makes $T_C = 8$. But at this moment there are still 4 more replies to get from the initial batch. After they are received, we have again $T_C = 4$ and so the client is allowed to make *another refill*, sending 4 new requests (*Q2*). The idle period follows, of length $T_0 - 4 \cdot T_{rpl}$. Then at moment *A1* we start getting the answers for *Q1*. After they arrive (*Q1* had only for 4 requests), the client touches again the L_W and refill *Q3* takes place. But the burst is not finished

because the replies for *Q2* start to arrive at *A2*. There is no spacing between the end of *A1* and the beginning of *A2* because the distance between the first request and the first reply in a refill cycle is constant ($T_0 - L_W \cdot T_{rpl}$). The end of *A2* coincides with refill *Q4* and then another idle period.

The difference between these two examples is in the number of refills done by the client during the period of one cycle. In the second example there are two refills and so the *effective burst length* is longer. We have in fact *two* back-to-back bursts of length $H_W - L_W$. The number of such *mini-bursts* is equal to the number of refills (a refill taking place whenever $T_C \leq L_W$). During the period of a cycle with H_W replies, we have $\left\lfloor \frac{H_W}{H_W - L_W} \right\rfloor$ refills. Therefore the effective burst length is:

$$\text{Burst Length} = B_L\left(L_W, H_W\right) = \left(H_W - L_W\right) \cdot \left\lfloor \frac{H_W}{H_W - L_W} \right\rfloor \tag{4.9}$$

4.4.3 Complete expression

Given the above corrections, we can express now the expected input rate in a network with delay T_0 and for a client configured with (L_W, H_W) as watermarks:

$$Rx(L_W, H_W; S_{rpl}, T_{rpl}, T_0) = \frac{\left(H_W - L_W\right) \cdot \left\lfloor \frac{H_W}{H_W - L_W} \right\rfloor \cdot S_{rpl}}{R\left(T_0 - L_W \cdot T_{rpl}\right) + \left(H_W - L_W\right) \cdot \left\lfloor \frac{H_W}{H_W - L_W} \right\rfloor \cdot T_{rpl}} \tag{4.10}$$

In the next section we present a set of measurements and compare them to the expected values.

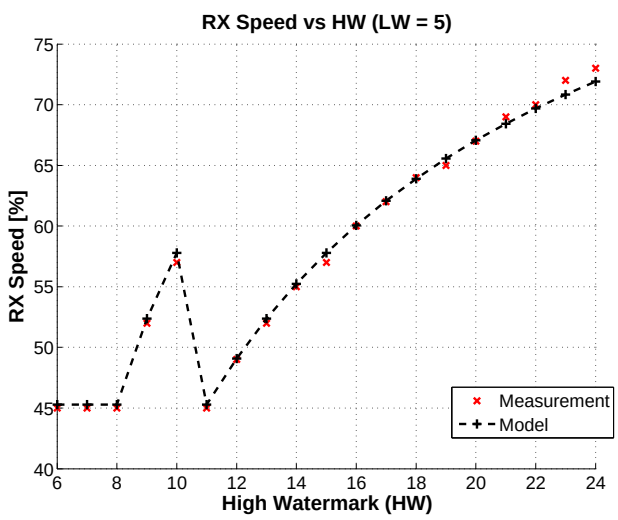
4.5 Measurements using the GETB hardware

4.5.1 Input rate

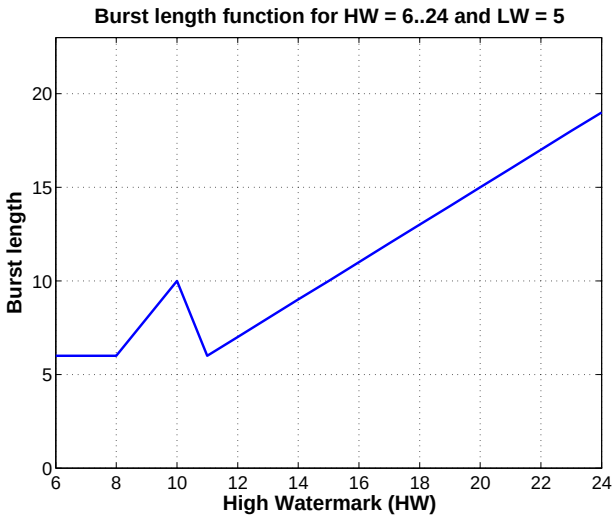
The results from this section were obtained using the GETB tester. The network emulator was used in some cases to increase the delay in the network and to modify in this way the network capacity (via the delays included in T_0). We plotted the measured and predicted *RX Speed* values versus the watermarks.

4.5.1.1 L_W = const, H_W = variable

The first set of results was obtained by configuring the client node with a fixed value for L_W and then varying H_W. The only condition imposed is $L_W < H_W$.



(a) *RX Speed* vs HW (LW=5, HW=6..24)



(b) Burst length function for LW=5, HW=6..24

Figure 4.7: RX Speed measurements: L_W =const, H_W =variable.

As it can be seen from Figure 4.7(a), the *RX Speed* increases according to the prediction[25]. The growth of the rate is only due to the longer bursts of replies. The *RX Speed* reaches 100% only if L_W is larger than the network capacity, N_C (not the case here). The "bump" that can be observed is due to the burst length function (Figure 4.7(b)).

4.5.1.2 L_W = variable, H_W = const.

The effects of the variable burst length become more visible when H_W is kept constant and we change only L_W. Figure 4.8 contains a few examples. The network capacity was 12, so the maximum speed can only be reached for $L_W > 12$. The figure illustrates that all the experimental curves match very well the theoretical predictions.

When H_W is exactly equal to the network capacity, $H_W = N_C$, then L_W has no influence on the *RX Speed*, until it reaches the value $\frac{H_W}{2} = \frac{N_C}{2}$. As long as $L_W < \frac{N_C}{2}$, the *RX Speed* is equal to half of line speed:

> When $L_W \in [0, \frac{H_W}{2}]$ we have the Burst Length $= H_W - L_W = N_C - L_W$. The denominator of (4.10) becomes[26]: $NC \cdot T_{rpl} - L_W \cdot T_{rpl} + (N_C - L_W) \cdot T_{rpl} = 2 \cdot (N_C - L_W) \cdot T_{rpl}$. So the expression for *RX Speed* will finally be: $\frac{(N_C - L_W) \cdot S_{rpl}}{2 \cdot (N_C - L_W) \cdot T_{rpl}} = 50\%$ line-speed. The second set of curves ($H_W = 12$) in Figure 4.8 shows that this is indeed the case.

[25] Please note that the origin of the X-Y axes has been suppressed in some of the plots presented in this chapter. This was done in order to highlight the dynamic range of the displayed quantities.

[26] We used the fact that $T_0 = N_C \cdot T_{rpl}$.

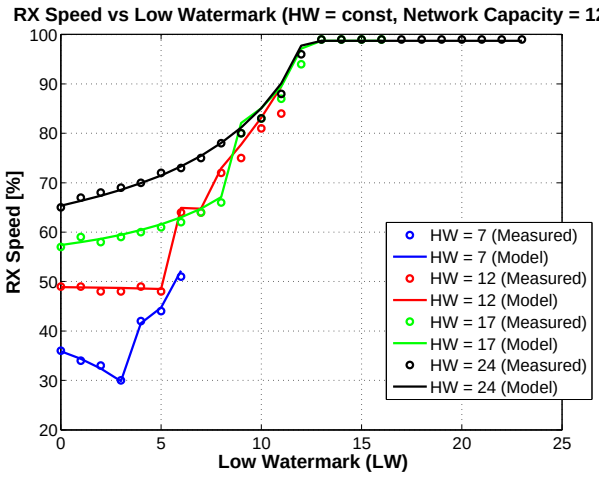


Figure 4.8: RX Speed measurements: L_W =variable, H_W =const..

4.5.1.3 $L_W = H_W - 1$

If the low watermark follows closely the high watermark, i.e. if $L_W = H_W - 1$, it means that the client is "aggressive", sending a new request as soon as it gets a reply. The new expression for *RX Speed* will be linear[27], as shown in (4.11). The dependency is confirmed by measurements (Figure 4.9) – the curves show that for small network capacities, the *RX Speed* reaches line-speed very fast (the first curve).

$$\begin{aligned} Rx &\overset{(4.10)}{=} \frac{H_W \cdot S_{rpl}}{R(T_0 - (H_W - 1) \cdot T_{rpl}) + H_W \cdot T_{rpl}} \\ &= \begin{cases} \frac{S_{rpl}}{T_{rpl}} \overset{(4.4)}{=} 100\% & \text{if } H_W - 1 \geq N_C \\ \underbrace{\frac{S_{rpl}}{T_0 + T_{rpl}}}_{const.} \cdot H_W & \text{if } H_W - 1 < N_C \end{cases} \end{aligned} \tag{4.11}$$

4.5.1.4 L_W = variable, H_W = variable

Varying both watermarks (all possible combinations in the range 30×30), we obtained the surface shown in Figure 4.10(a). The network capacity was 11; the surface generated by formula (4.10) is almost identical to the measurements. Cross sections through this surface are shown in Figures 4.10(b) and (c). The relative error[28] between the measured *RX Speed* surface and the predicted one is shown in Figure 4.11 – we represented the 3D surface (a) and a 2D color map

[27]The expression is linear before it saturates to the line speed.

[28]For a value a and its approximation, b, the relative error is given by $\frac{|a-b|}{|a|} \times 100\%$.

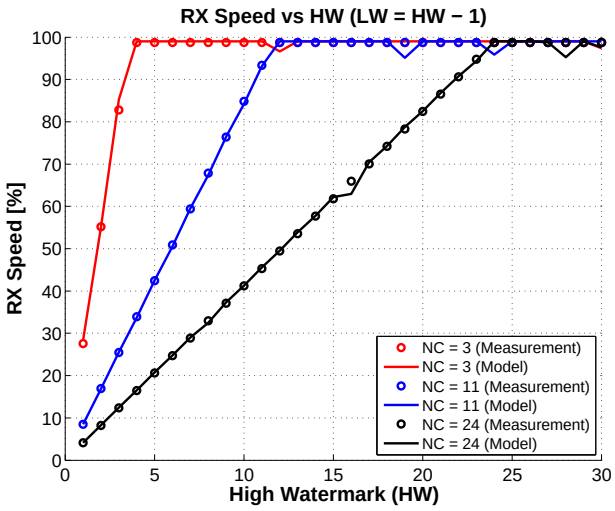
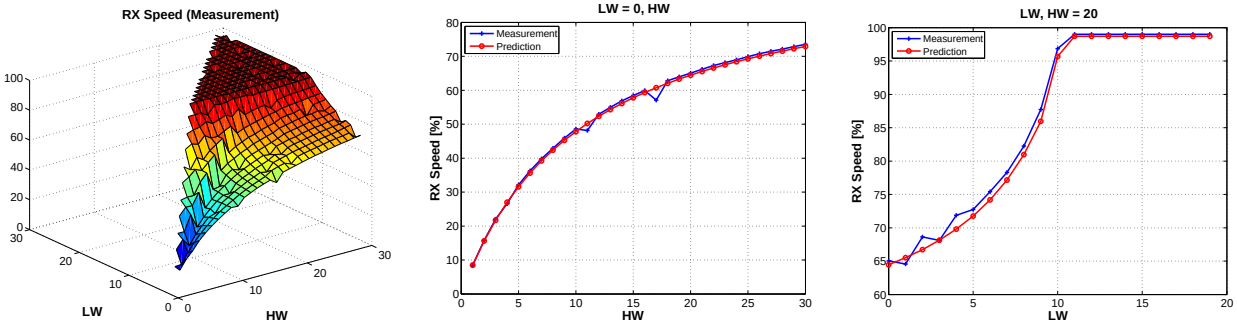


Figure 4.9: RX Speed measurements: L_W =H_W-1.

(b). The histogram of the values is visible in Figure 4.11(c). The maximum error is 6%; more than 40% of the measured values have errors less than 1%.



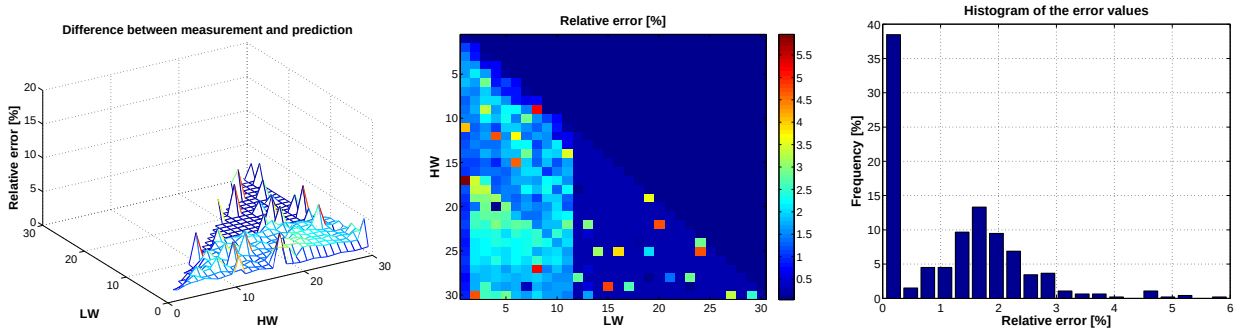
(a) Measurement (b) $L_W = 0$, H_W = var (c) L_W = var, $H_W = 20$

Figure 4.10: RX Speed measurements: 3D surface and cross sections (N_C= 11).

4.5.2 Multiple servers

The analysis carried out until now did not consider the number of servers that a client talks to. Because the *RX Speed* is determined only by the tokens, the measurements are insensitive to the number of servers. As a matter of fact, the results from Figure 4.10 were obtained in a setup with one client and 5 servers.

The difference between single and multiple servers, from the client's point of view, appears in the round-trip (response) time. The *Round-Trip Time* (RTT) can be defined as the sum between



(a) 3D surface (b) 2D color map (c) Histogram

Figure 4.11: RX Speed measurements: Relative error between measurement and prediction.

the one-way delay of the request plus the time spent inside the server plus the one-way delay of the reply (also called *response delay* or *reply latency*). When we use multiple servers, the one-way delay of the replies increases (and so does the RTT). The servers respond almost simultaneously to the requests of the same client and this generates a temporary congestion in the network, at the output port of the switch, towards the client. As the replies queue in the buffer of the switch, their delay is proportional to the usage of this buffer.

The effect can be observed in Figure 4.12. Here, we measured the *RX Speed* and the one-way delay of responses (a component of the RTT) during a test like in Section 4.5.1.2 – we kept $H_W = 24$ and varied L_W. We ran it first with one server and then with 5 servers. The *RX Speed* curves are similar in the two cases, but the response latencies differ.

For a single server, the response latency is flat[29], for multiple servers, it varies. The V-shape variation will be explained in the next section. Now we just point out that the shape of the response delay curve depends on the amount of data that accumulates in the (output) queue of the switch. The queue occupancy in a switch cannot be accurately measured unless special support is provided from the hardware manufacturer. The RTT or the individual response delay can be used to estimate it[30].

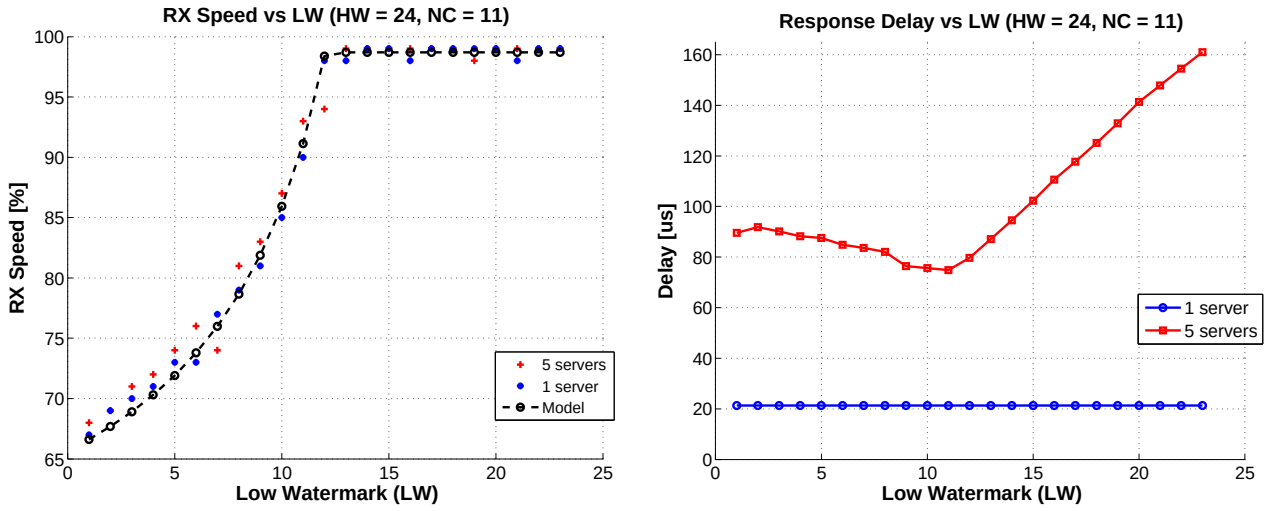
When the client talks to a single server, we've seen that the response delay remains constant – the dependency on watermarks appears at the level of the internal queue of requests, inside the server. The length of this queue can be measured precisely in our test-bed. We assume the following two dependencies are equivalent:

- Server request queue occupancy[31] (number of requests), as a function of (L_W, H_W).

[29]Please note that we measured only the delay from the server to the client, not the entire Round-Trip Time. When there is a single server, requests wait in the internal queue of the server. This means that the RTT will still have a variable component. The one-way delay of responses will be constant (because the network path is free), but the complete RTT will include a variable time spent inside the server.

[30]The current buffer occupancy can be found if the maximum value for the RTT is known (which corresponds to a full queue/buffer). More details are available in Section 4.8.

[31]The server queue occupancy is measured with the GETB tester.



(a) *RX Speed*

(b) Response Delay

Figure 4.12: Multiple servers – RX Speed and the Response Delay.

- Switch output port queue occupancy[32] (number of replies), as a function of (L_W, H_W).

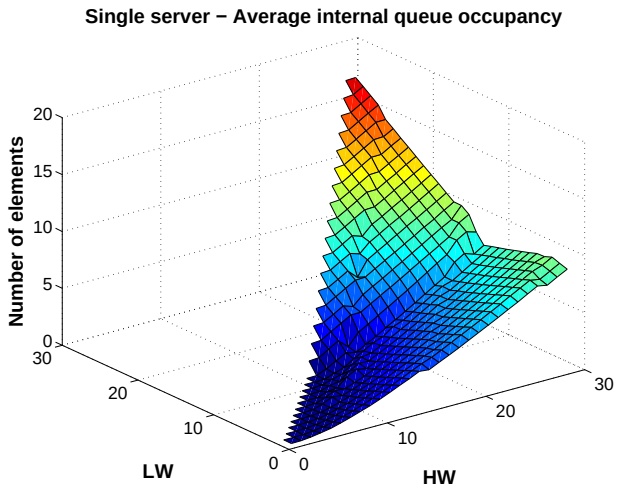
To support this statement we ran a scan of all possible watermark combinations in the range 30×30 (we kept the same settings as for the measurements described in Figure 4.10). The first test was done in a 1 to 1 setup, the second in a 1 to 5 setup. In both cases we recorded the response delay and internal queue occupancy at the servers.

Figure 4.13 shows the plots for the internal queue. The surface from Figure 4.13(a) (for the single server) will be studied in the next section. With multiple servers, the internal queue remains almost empty because the servers share the load (Figure 4.13(b)). Note that 2 is the maximum value on the Z-axis. As long as $H_W < 5$, the queue remains zero because the five servers can absorb all the requests. When $H_W > 5$, the average queue occupancy increases also with multiple servers. The queue occupancy surface has "isolines" for $H_W - L_W = \text{const}$.

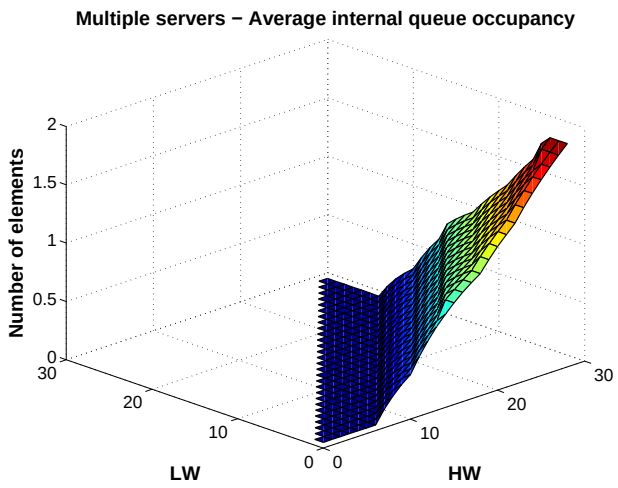
Figure 4.14 presents the average delay of the responses, during the same tests – this time the figures (a) and (b) seem to be reversed if we compare to Figure 4.13. We remark the similarity of figures 4.13(a) and 4.14(b). For multiple servers, the delay dependency looks like the internal queue dependency for a single server. The response delay shown in Figure 4.14(b) is proportional to the buffer utilization at the output port of the switch. The delay is caused by replies that accumulate in the switch output buffer. Each reply is transferred in T_{rpl} seconds. Therefore this should be the proportionality constant between the observed delay and the actual buffer occupancy.

We "divided" the delay surface from Figure 4.14(b) by the factor T_{rpl}. From the result we subtracted the internal queue occupancy surface from Figure 4.13(a). The difference surface

[32]The output port queue occupancy is estimated using the RTT / response delay.

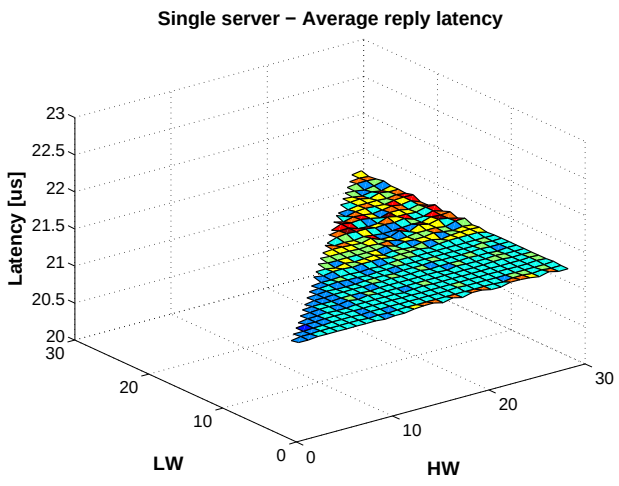


(a) Single server

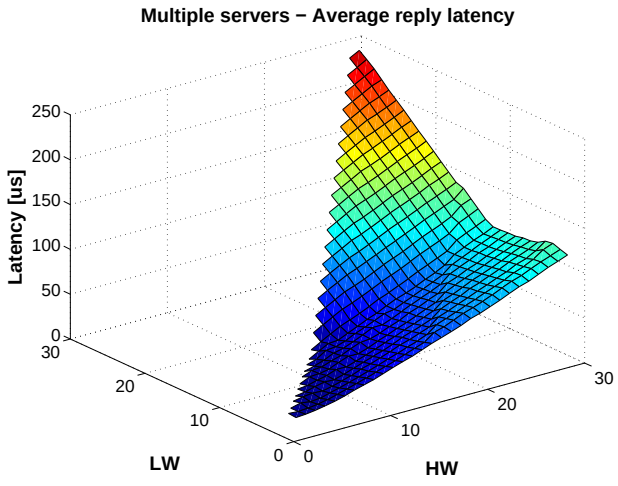


(b) Multiple servers

Figure 4.13: Comparison – Single and multiple servers (N_C=11) – Internal queue occupancy.

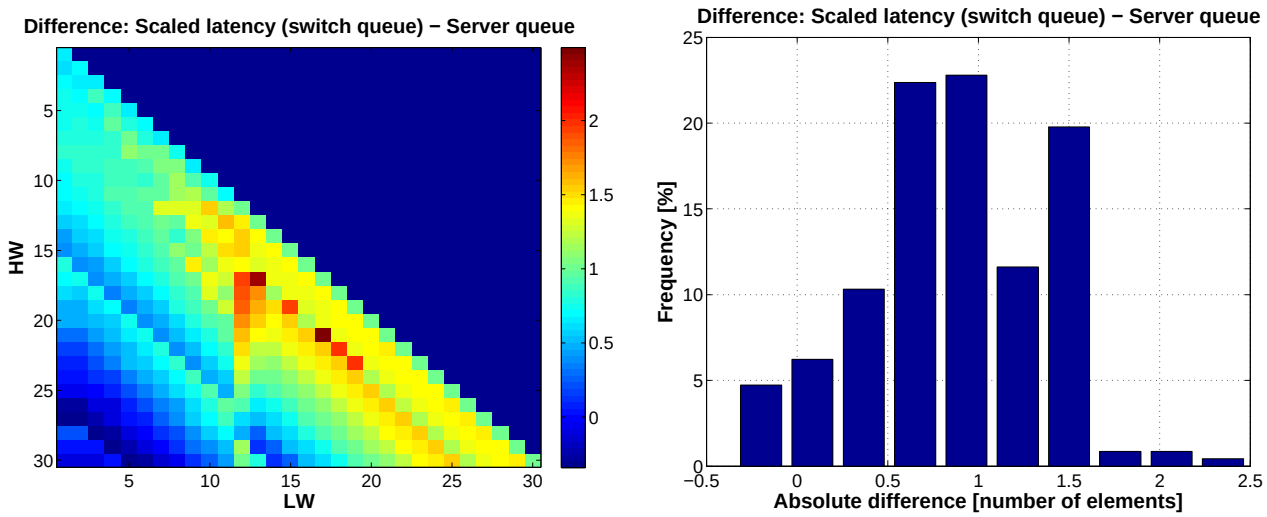


(a) Single server



(b) Multiple servers

Figure 4.14: Comparison – Single and multiple servers (N_C=11) – Response delay.



(a) Difference image

(b) Histogram of the difference values

Figure 4.15: Comparison – Difference between switch queue and server queue.

appears in Figure 4.15. On side (a) the surface is represented as a color map and on side (b) we plotted the histogram of all values of the difference. We observe that, on the average, the difference is very small (about one unit, i.e. one element in the queue).

These results indicate that the queue in the switch and the one in the server depend on the watermarks in the same way. This assumption is used in the next section, where we study queueing at the level of a server and we extrapolate the results to the level of a switch.

4.6 Queueing in a request-response system

Queues are generally provided to overcome *temporary* congestion. Due to the funnel-shaped traffic pattern, in ATLAS it is more likely to have congestion, hence significant buffering, on the reply path, from the servers to the clients.

We summarize below the places where we can expect queues to build up in a request-response system with multiple clients and servers (considering the assumptions from page 84, Section 4.4.1).

- *The buffers of the switch.* Packets that cannot by transmitted (because the output line is full) are kept by the switch into its own buffer memory. This can happen at the output port of a switch, for a port that connects to a client node[33].

[33]It is also possible to buffer packets at the input ports, when they enter the device. This technique is not generally used because it leads to *Head of Line Blocking*, i.e. packets going to un-congested ports have to wait in the queue behind packets destined to oversubscribed ports.

- *The internal queue of the servers.* A server that cannot respond fast enough (typically due to the lack of network bandwidth[34]) will put the incoming requests on hold into an internal queue. This should not happen if the number of clients is less than the number of servers, so the output rate of a server is always less than line speed[35]. It can happen if the switches "propagate" the congestion from an output port, back to the sources contributing to it. This is called *flow-control* and is meant to slow down the transmitters (servers in this case). The method is not reliable because most operating systems do not inform the applications about the lack of network bandwidth, so the servers will not reduce their speed.

- *The queues of the operating system.* The operating system may buffer data before delivering it to the application. In a multitasking environment this is important because the CPU is shared between many applications. The OS will buffer any data until an application is scheduled for a slice of CPU time.

- *The queues of the network card.* A network card has a small amount of memory to hold a few packets which cannot be read by the host machine (on the ingress, input side). The operating system is responsible to empty the buffers of the network card.

From this list, the first two are the most likely candidates for queueing. Based on the assumption made at the end of Section 4.5.2, that we have the same dynamics for internal queues in servers and for the switch buffers, we shall present in the following results based on the queues measured inside servers.

Before we go forward with the measurements, we shall add a few words about N_C, the network capacity introduced in Section 4.4.2.1. Its value equals the number of replies that can "move" through the network at a given moment. We remind that the high watermark, H_W, is the *maximum* number of transactions[36] that are active in the network. The low watermark, L_W, represents the number of transactions that we would like to be "in transit", i.e. to be moving towards the client. The network capacity represents the maximum number of transactions allowed by the *network* to be in transit. Anything above this limit will have to be temporarily queued. Note that the network capacity is directly proportional to the network delay, which does not have to be a constant – it can change in time due to other factors[37]. In order to highlight the effects of the network capacity, we used the GETB network emulator to increase the delays between the servers and the clients.

[34]The GETB and *mpNetPerf* will be limited by the network bandwidth. The real ATLAS applications, the ROS for example, may be slowed down due to other factors not related to the network.

[35]Assuming an uniform distribution of the requests among the servers. Again, this might not happen in the real ATLAS. Some servers may receive more requests than others. This depends on the Regions of Interest determined by the Level 1 trigger.

[36]Each request which generates a response can be considered a *transaction*. The transaction is "active" from the moment the client issues the request until the corresponding reply is received. A transaction can be either "pending" i.e. the reply not yet generated, or "in transit" meaning that the reply is on its way back to the client.

[37]Network congestion or scarce CPU resources are examples of factors that can modify the end-to-end network delay.

4.6.1 Measurements and explanations

We used the GETB test-bed to study the dependency on watermarks of the internal queue of the server (in a one-to-one scenario). The capture mode of the GETB (Section 3.3.1.4) allowed us to record the state (occupancy) of the queue in the server, after each request received.

For most of the tests, we recorded the minimum, average and maximum occupancy of the internal queue. The maximum occupancy was saved after the startup of the test, in order to avoid recording the initial burst which is always equal to the high watermark (we are more interested in the "steady state" values).

4.6.1.1 L_W = const, H_W = variable

We begin with the simplest case, when $L_W = 0$. As the client always sends bursts of H_W requests, the maximum queue occupancy is always H_W [38]. The client waits for all the replies from the server, so the minimum occupancy is zero.

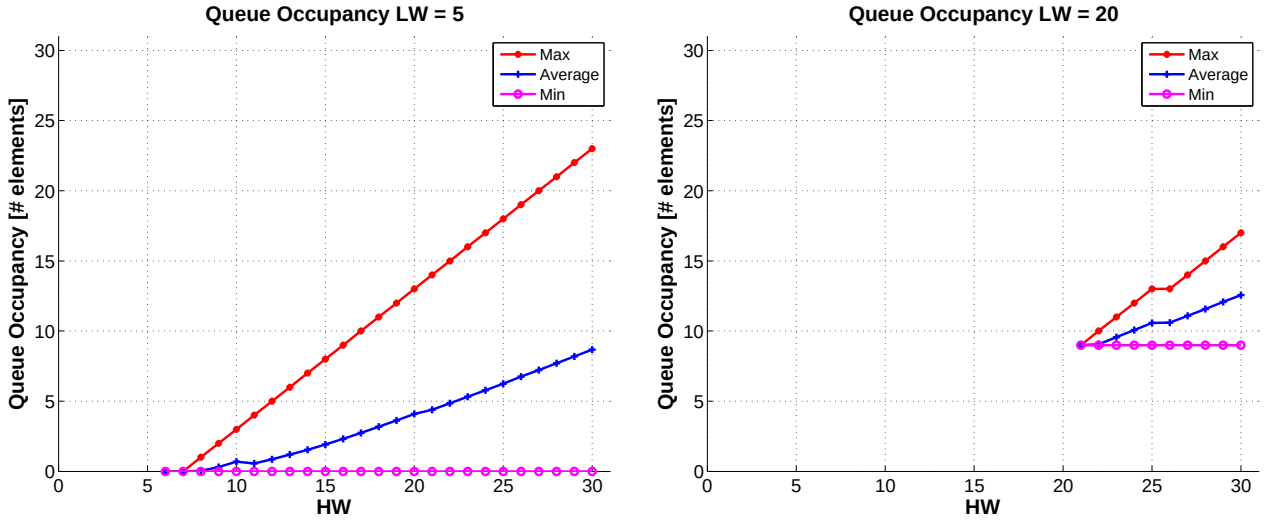
When $L_W > 0$, the system is never "idle", i.e. there are always L_W replies or requests in transit. We distinguish two cases: $L_W < N_C$ and $L_W \geq N_C$.

1. If $L_W < N_C$ then there are periods of "silence" in the network ("gaps" in the data stream). The internal queue of the server gets empty for short periods, while replies are moving towards the client and this one has not reached yet the low watermark. The queue occupancy will vary between zero and an upper limit (Figure 4.16(a)). After the initial burst of H_W requests, the client sends only bursts of $H_W - L_W$ requests to the server. So the upper limit for the queue is $H_W - L_W$. Figure 4.16(a) shows the results for this case (in good agreement with the predictions).

2. If $L_W > N_C$, we have a server trying to send data faster than line-speed. As this is impossible, it has to queue requests and we shall see an upward shift of the minimum queue size – the offset value corresponds to the excess above the network capacity: $L_W - N_C$. If in the first case the maximum queue occupancy was growing as $H_W - L_W$, now the server has to absorb not only $H_W - L_W$ requests each time, but also what is over the capacity: $L_W - N_C$. So the maximum occupancy will be $H_W - L_W + L_W - N_C = H_W - N_C$. Plots reflecting the second case are shown in Figure 4.16(b).

The average queue occupancy We'll explain now how we estimate the average queue occupancy. Using the packet capture mode, we obtained Figure 4.17(a) which shows the evolution in time of the number of elements in the server request queue. For this example, the requester client was configured with $L_W = 0$ and $H_W = 30$.

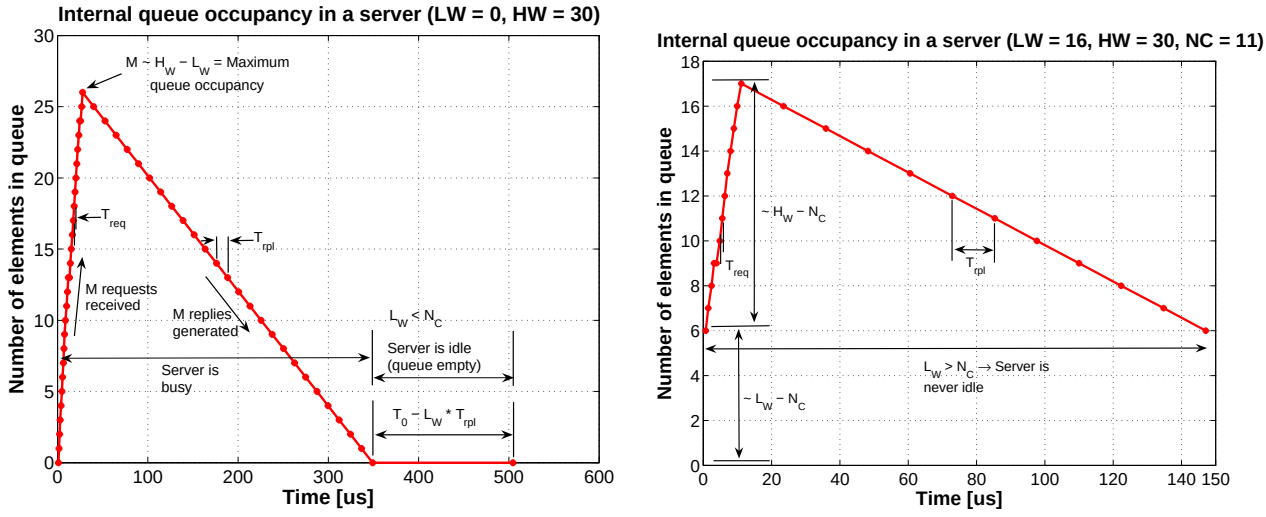
[38] In the case of the GETB, the maximum occupancy is less than H_W because the system contains other queues which cannot be inspected, i.e. requests or replies are buffered in other places, different from the internal queue in the server. However, these queues are small – their estimated cumulative capacity is of 3 or 4 elements.



(a) $L_W < N_C$ (L_W =5,N_C=11)

(b) $L_W > N_C$ (L_W =20,N_C=11)

Figure 4.16: Queueing: L_W =const, H_W =var..



(a) $L_W = 0$,$H_W = 30$, $L_W < N_C$

(b) $L_W = 16$, $H_W = 30$, $L_W > N_C$

Figure 4.17: Queue development in time.

The snapshot contains one complete "refill cycle". It begins with the accumulation of requests from the client. This is the initial ramp up of the number of elements in the queue. The queue occupancy grows up to a maximum value M. When $L_W = 0$, M should be approximatively equal to H_W. If $L_W > 0$ then the maximum decreases with the value of L_W as explained in the previous section: $M = Q_{max} = H_W - L_W$.

After the initial bursts of requests, which takes $M \cdot T_{req}$ seconds, the queue is emptied, by sending back the replies. This process needs $M \cdot T_{rpl}$ seconds to complete. After that there is a period of "silence" while the queue is empty. In the example shown, the length of the silence period is maximum and corresponds to T_0, the Base Round-Trip Time. When $L_W > 0$, the length of this period decreases with $L_W \cdot T_{rpl}$ as explained in Section 4.4.

We denote by q_i the occupancy at instant i. This value changes after w_i seconds. During the initial ramp up we have $w_i = T_{req}$ and then $w_i = T_{rpl}$ when the queue is emptied. Let $T_{tr} = T_{req} + T_{rpl}$. With these notations, we have:

$$\begin{aligned} Q_{avg} &\overset{L_W \le N_C}{=} \frac{\sum_i q_i \cdot w_i}{T_{cycle}} = \frac{\sum_{i=0}^{Q_{max}} i \cdot T_{req} + \sum_{0}^{i=Q_{max}} i \cdot T_{rpl} + 0 \cdot T_{empty}}{Q_{max} \cdot T_{req} + Q_{max} \cdot T_{rpl} + T_{empty}} \\ &= \frac{T_{tr} \cdot \frac{Q_{max} \cdot (Q_{max}+1)}{2}}{Q_{max} \cdot T_{tr} + R(T_0 - L_W \cdot T_{rpl})} \end{aligned} \tag{4.12}$$

The expression (4.12) is valid for $L_W < N_C$ (Figure 4.17(a)). In Figure 4.17(b) we show what happens if $L_W > N_C$. In this case the maximum $M = Q_{max} = H_W - N_C$ and the minimum will be $Q_{min} = L_W - N_C$. There are no "silent" periods. The average will be given by:

$$\begin{aligned} Q_{avg} &\overset{L_W \ge N_C}{=} \frac{\sum_{i=Q_{min}}^{Q_{max}} i \cdot T_{req} + \sum_{Q_{min}}^{i=Q_{max}} i \cdot T_{rpl}}{(Q_{max} - Q_{min} + 1) \cdot T_{req} + (Q_{max} - Q_{min} + 1) \cdot T_{rpl}} \\ &= \frac{Q_{max} + Q_{min}}{2} = \frac{H_W + L_W}{2} - N_C \end{aligned} \tag{4.13}$$

4.6.1.2 L_W = variable, H_W = const.

The dependencies on the other parameter, L_W, can be seen in Figure 4.18. The breakpoint in the middle of the plot corresponds to $L_W = N_C$.

The initial downward slope of the queue utilization was explained in the previous section: as L_W grows, the client "refills" the server with $H_W - L_W$ requests, so the queue size at the server shrinks (because H_W is constant). By increasing L_W we use more and more of the network

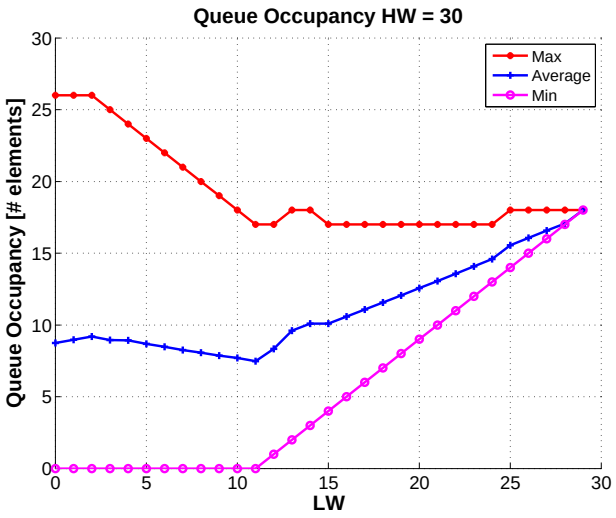


Figure 4.18: Queueing: L_W =var, H_W =const.

capacity and less resources on the server. When the network capacity is reached, a queue starts to build up inside the server, because the network cannot transfer replies faster. The minimum queue occupancy is proportional to the difference $L_W - N_C$. The maximum queue occupancy remains constant at $H_W - N_C$ because H_W does not change. The average occupancy is given by (4.12) and (4.13). We remark that this case explains the V-shape from the example on page 94, Figure 4.12(b).

4.6.2 Expressions for the queue occupancy

Given the above observations, we can synthesize the following expressions for the minimum, maximum and average queue occupancies. These expressions were verified for the internal queue of a server; experimental results (Section 4.5.2) show that they apply also when the buffering is inside the network.

Minimum

$$Q_{min} = R(L_W - N_C) \tag{4.14}$$

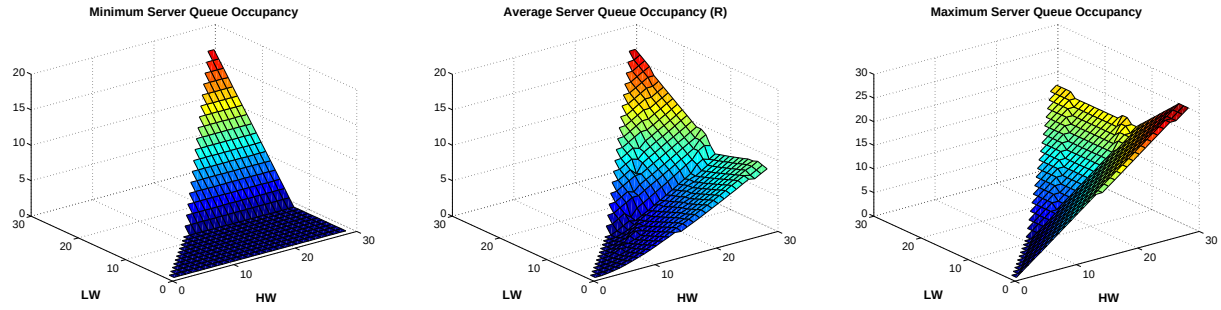
Maximum

$$Q_{max} = \begin{cases} H_W - L_W & \text{if } L_W < N_C \\ H_W - N_C & \text{if } L_W \geq N_C \end{cases} = H_W - \min(L_W, N_C) \tag{4.15}$$

Average

$$Q_{avg} = \begin{cases} \frac{T_{tr} \cdot \frac{(H_W - L_W) \cdot (H_W - L_W + 1)}{2}}{(H_W - L_W) \cdot T_{tr} + R(T_0 - L_W \cdot T_{rpl})} & \text{if } L_W < N_C \\ \frac{H_W + L_W}{2} - N_C & \text{if } L_W \geq N_C \end{cases} \tag{4.16}$$

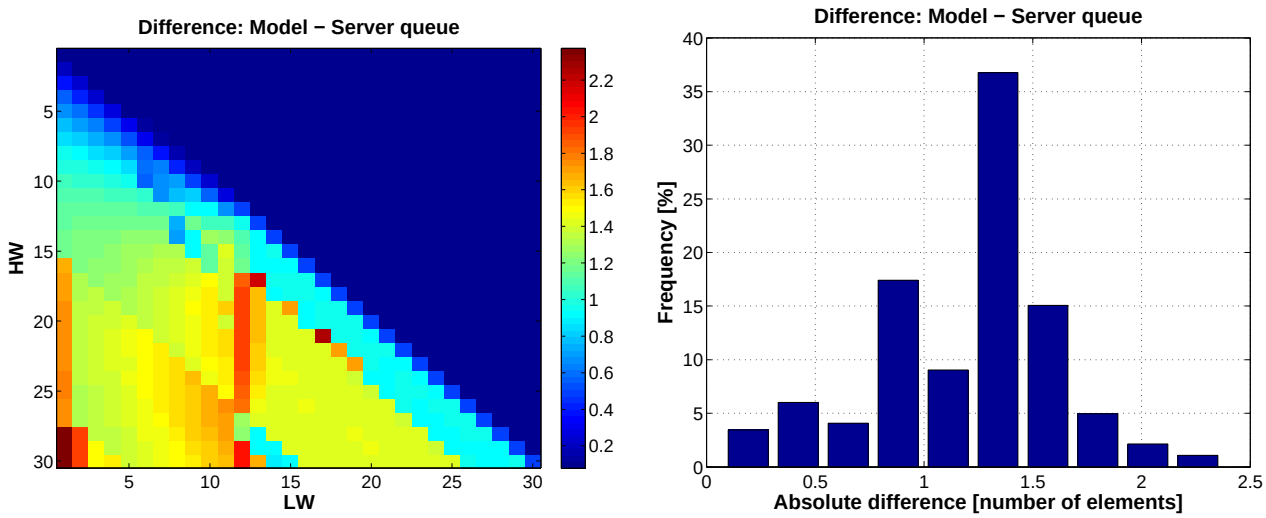
In Figure 4.19 we show the results of the internal queue measurement for all valid combinations of watermarks. The surfaces are correctly described by the above expressions.



(a) Minimum (b) Average (c) Maximum

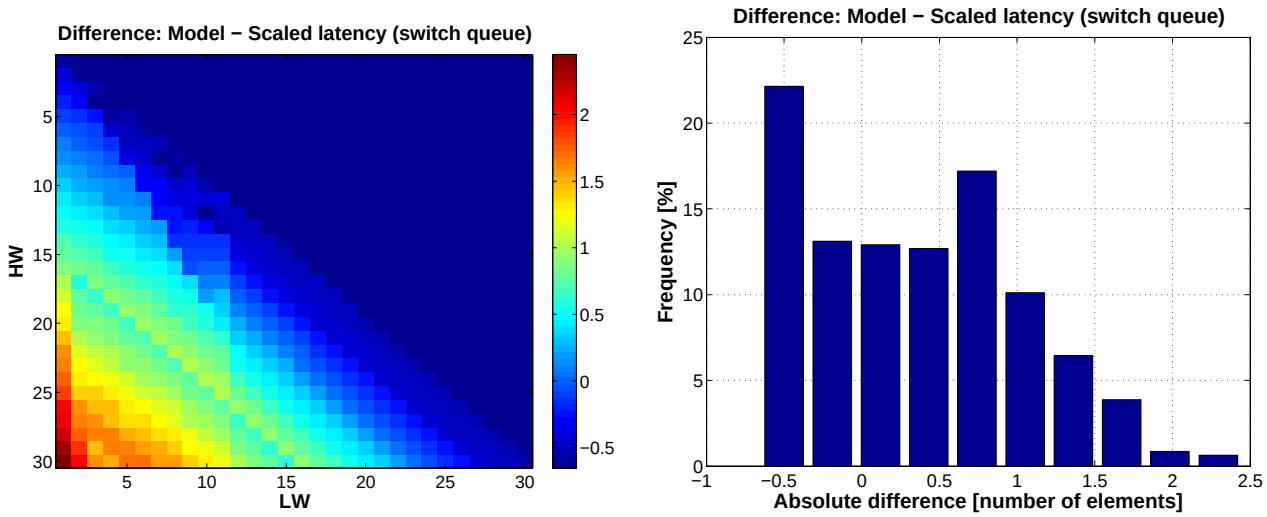
Figure 4.19: Queueing: Server's internal queue occupancy (3D plots).

We compared the surface generated by the formula for the average occupancy (4.16) with the measurements presented in Section 4.5.2 (see Figures 4.13(a) and 4.14(b)). The differences images and the corresponding histogram of all difference values are shown in Figures 4.20 and 4.21. We observe an average difference of at most one element between our model and the experimental results.



(a) Difference image (b) Histogram of the values

Figure 4.20: Difference between queue model and server queue (see Figure 4.13(a)).



(a) Difference image

(b) Histogram of the values

Figure 4.21: Difference between queue model and switch port buffer (see Figure 4.14(b)).

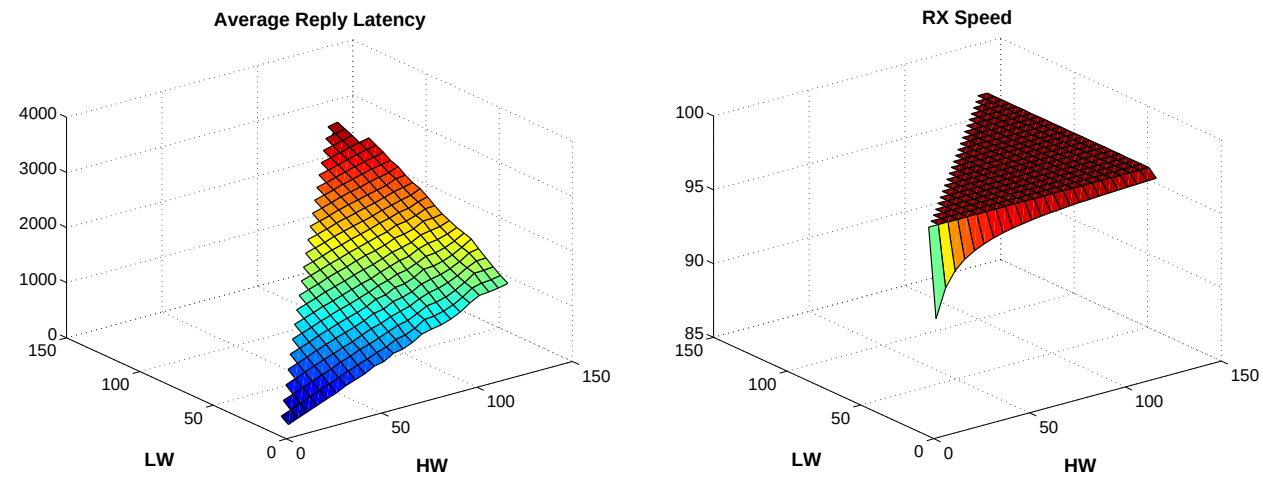
4.6.3 Queueing in a switch

In practice, when the network consists of just a few switches, the delays are so small that N_C is of the order of just a few packets. In this case the average queue occupancy is approximatively $\frac{H_W+L_W}{2}$ according to (4.16). When multiple servers are used then a queue grows at the switch buffer level and the response delay should also be proportional to $\frac{H_W+L_W}{2}$.

Latency as a measure of the queue occupancy For a switch with buffers that can accommodate about 125 replies, we measured the average reply latency during a "scan" for all watermarks up to 125. We expect the surface from Figure 4.22(a) to be proportional (by a factor k[39]) to $\frac{H_W+L_W}{2}$. In order to check this, we divided this surface by the one given by $Z(H_W, L_W) = \frac{H_W+L_W}{2}$. The result is shown in Figure 4.23 as an image map and as a histogram of the values. We observe that the proportionality coefficient k is not constant, but its standard deviation σ is small compared to the average ($100 \cdot \frac{\sigma}{\mu} = 6\%$).

The *RX Speed* is shown in Figure 4.22(b) – it reaches immediately line-speed because the network capacity is almost zero. As long as the *RX Speed* stays at 100%, the increase in watermarks translates only into additional queueing delay. Packet loss appears when the average of the watermarks is above the buffering limit.

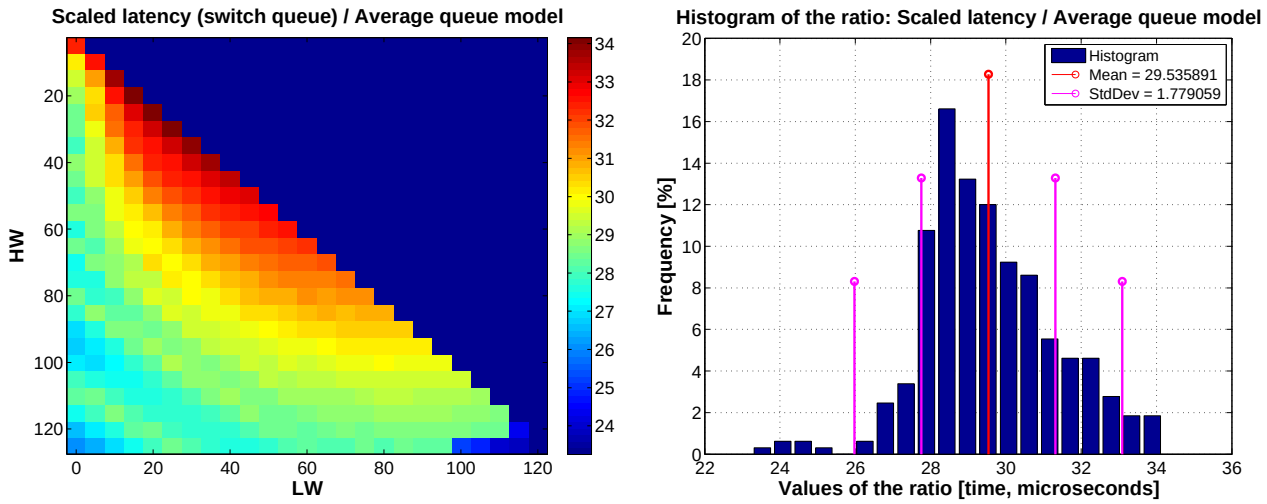
[39]The value of k depends on the size of the reply and on its transfer time. The test in this example used replies composed of 3 packets of 1512 bytes. Their total transfer time is $T_{rpl} = 3 \cdot 12.25\mu s = 36.75\mu s$.



(a) Response delay

(b) *RX Speed*

Figure 4.22: Queueing: Filling the output buffer of a switch (GETB).



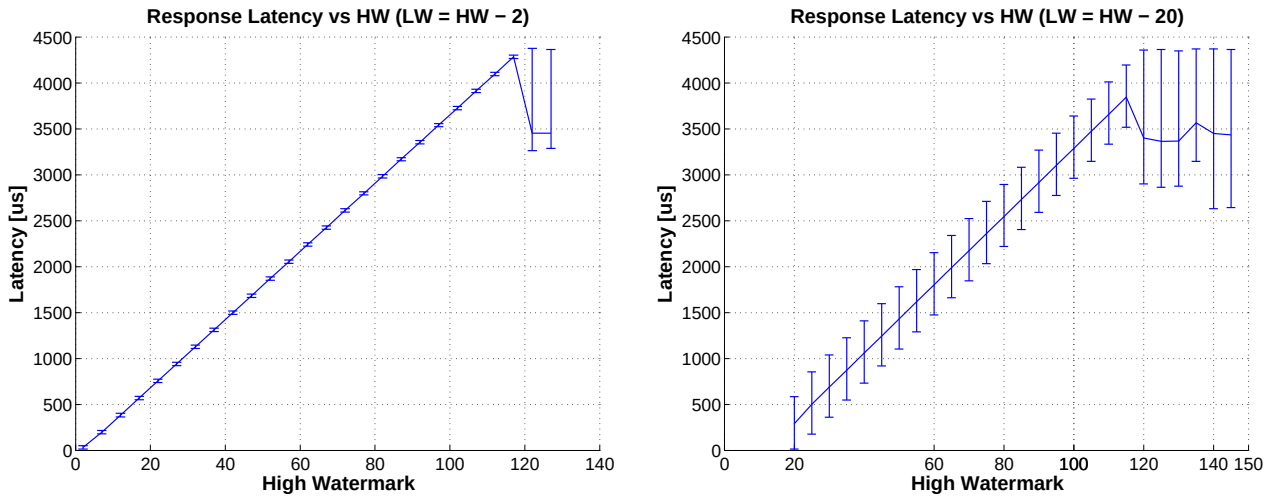
(a) Image map of the ratio

(b) Histogram of the values

Figure 4.23: Proportionality factor: Latency in switch / Average queue model.

The watermark difference and the variation of the queue occupancy When the network capacity is negligible, the two watermarks determine the minimum and maximum queue occupancy. We show now a result that proves that the difference, D, in watermarks really determines the amplitude in the variation of the queue occupancy. During a test like the one described above, we've recorded the latencies for the test runs with watermarks given by $D = H_W - L_W = 2$ and $D = H_W - L_W = 20$.

Figure 4.24 shows the results: the error bars around the averages represent the variation[40] of the delays. The deviation is constant and proportional to the difference D as long as the switch buffer is not full. Towards the end of the axis, we observe that the maximum latency presents a plateau given by the buffer size. When the switch buffer is almost full, the switch can drop packets from the tail of the queue[41]. For efficiency reasons, it drops not one, but longer sequences of packets[42]. The packets that arrive immediately after, will find a queue which is less occupied. This explains the bend of the average latency towards slightly lower values (at the end of the X-axis). If the difference between the watermarks is small and constant (one unit for example), we can impose an (almost) fixed value for the queue occupancy (Figure 4.24(a)). This observation will be used as a measurement tool in Section 4.8.



(a) $H_W - L_W = 2$ (b) $H_W - L_W = 20$

Figure 4.24: Queueing: Dispersion of the response delay as a function of the difference in watermarks.

[40] Difference between minimum and maximum value.

[41] This technique is called *Random Early Discard* (RED). The probability of dropping packets increases with the average utilization of the queue. The RED algorithm is designed to improve congestion handling in a network.

[42] This statement is supported by the example from Section 3.3.1.4, page 65.

4.7 Measurements in software

In this section we discuss results obtained using the *mpNetPerf* software[43]. They are not identical, even if the program implements exactly the same traffic shaping method. We assume that the inaccuracies are mostly due to the sharing of the CPU and network resources (multi-tasking).

We saw from Section 4.4 that the network delay plays an important role in the expression of the *RX Speed*. On the PC, the end-to-end delay behaves like a random variable. Figure 4.25 shows a histogram of the Base Round-Trip Time, T_0, for the GETB and for *mpNetPerf*. The T_0 was measured in a one-to-one setup by setting $L_W = 0$ and $H_W = 1$. Its value was saved during 10000 request-response transactions. The *mpNetPerf* histogram is much wider than the GETB one[44]. The two tests were done on the same switch, so the difference of $\approx 170\ \mu s$ is only latency added by the operating system and the application code.

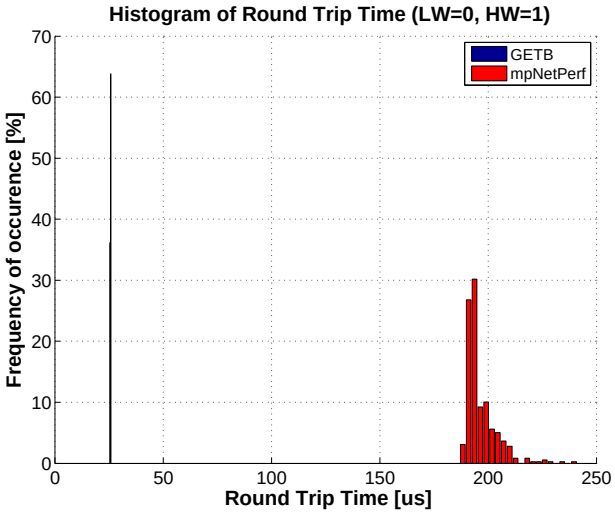


Figure 4.25: Histogram of the Base Round-Trip Time (T_0) – GETB vs mpNetPerf.

4.7.1 Input rate

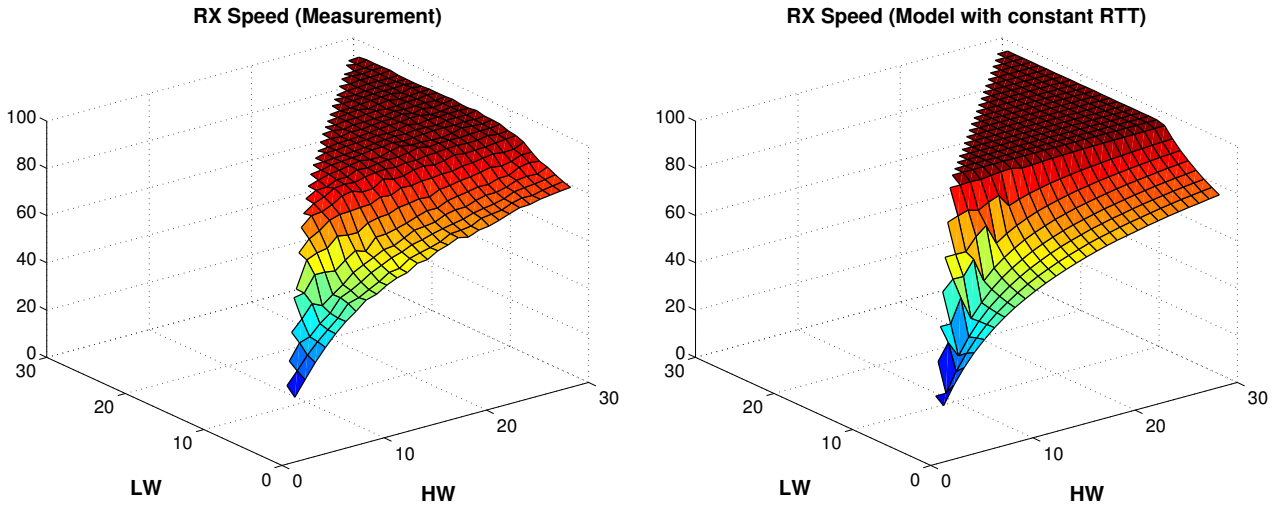
A watermark "scan", as in Section 4.5.1.4 has been tried using *mpNetPerf*, monitoring the *RX Speed* for each valid pair of watermarks. The tests were done with 1 client and 6 servers, using UDP[45]. We used up to 30 tokens, the same limit that was used for the GETB measurements. Figure 4.26 presents the *RX Speed* surface, compared to the expected values given by (4.10). Figure 4.27 contains two cross sections through the 3D surface. The T_0 used to compute the

[43]The measurements were performed on machines running Linux with kernel v2.4.21. Most of them had two Intel Xeon processors running at 2 GHz.

[44]The *mpNetPerf* data has: $\mu = 195\mu s$ and $\sigma = 13.56\mu s$. We have $100 \cdot \frac{\sigma}{\mu} = 6.9\%$. For the GETB case: $\mu = 25.6\mu s$, $\sigma = 0.15\mu s$, $100 \cdot \frac{\sigma}{\mu} = 0.6\%$.

[45]The ATLAS DAQ plans to use UDP for event data transfers. TCP is not used because it does not scale well with many parallel communication sessions.

predictions was the average value of the histogram from Figure 4.25 (so it includes the OS stack delays, in addition to the network latency).



(a) Measurement

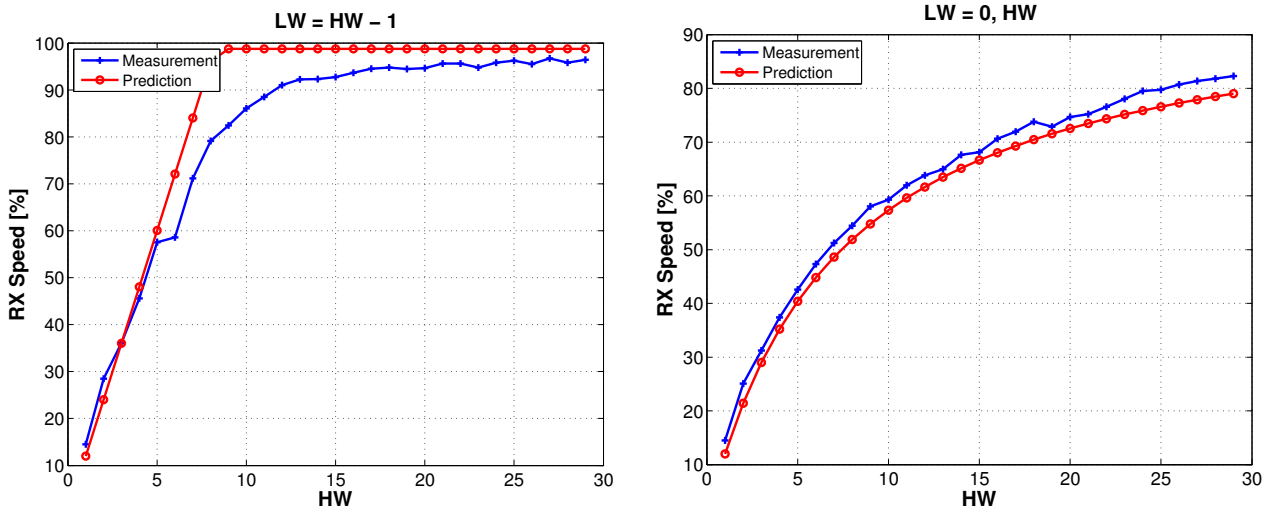
(b) Prediction

Figure 4.26: mpNetPerf: RX Speed measured with UDP.

We observe that the *RX Speed* formula (4.10) fits quite well the UDP curves/surface; the cross sections show a good match for the two commonly used configurations: $L_W = 0$ (L2PU) and $L_W = H_W - 1$ (SFI[46]). The similarity with the GETB can be explained by the fact that UDP traffic resembles more closely to what is generated by the GETB tester (raw Ethernet packets). The differences appear at higher loads because the host CPU is more busy with the network transfers. On the PC, the T_0 (a constant in formula (4.10)) is a function of the *RX Speed* itself.

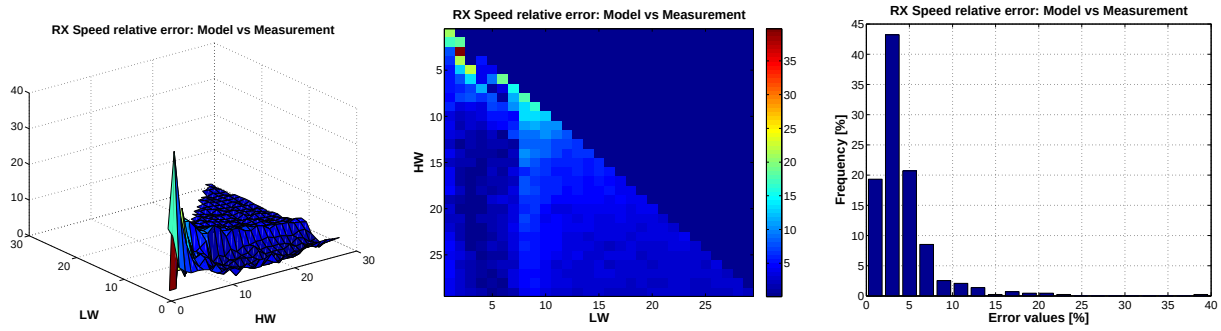
We show in Figure 4.28 the relative error surface with respect to the *RX Speed* model / formula. This figure can be compared to Figure 4.11. The errors are higher in this case, compared to the GETB, but the average relative error is still $\approx 3 - 4\%$. Part of the error is due to the fact that *mpNetPerf* never reaches exactly the full 100% line speed. In addition, the line rate is computed differently in *mpNetPerf* in comparison to the GETB. In *mpNetPerf* it is derived from the rate in bits per second of packet payloads. Because of the minimum interpacket gap on Ethernet[47], the effective line usage is higher than what is estimated by *mpNetPerf*. The GETB measures the real usage of the link, i.e. including the bandwidth "consumed" by the mandatory spacing between the packets.

[46]See Section 3.3.1.2 on page 63 for an explanation of why these settings are significant for ATLAS.
[47]For Gigabit Ethernet, it is the time required to send 80 bits of data or 0.16 μs.



(a) LW=HW-1 (b) LW=0

Figure 4.27: mpNetPerf: Cross sections through the RX Speed surface (UDP).



(a) 3D surface (b) 2D color map (c) Histogram

Figure 4.28: mpNetPerf: Relative error between RX Speed measurement and prediction.

4.7.1.1 Comparison with the TDAQ software

We've made a test with the TDAQ software in order to compare the results to those of *mpNetPerf*. We configured a small event-building setup, with one SFI (the client) and two ROS computers (servers). The SFI sends requests to the two ROSs; the number of outstanding requests is limited by TS, the *Traffic Shaping Parameter* of the SFI. This is equivalent to the High Watermark in our notations. As the SFI keeps a constant number of TS unserved requests, we can say that the Low Watermark is $L_W = H_W - 1$. The test was done with all three nodes connected to the same switch. We varied the TS parameter in the SFI and we measured the *RX Speed*. In the same hardware setup (same computers, same switch), we ran a similar test with *mpNetPerf* (with one client and two servers). The results are presented in Figure 4.29. The shape of the 'ATLAS' *RX Speed* curve is very similar to the one produced by *mpNetPerf*. At higher speeds, *mpNetPerf* has a slight advantage because of less overhead per transaction.

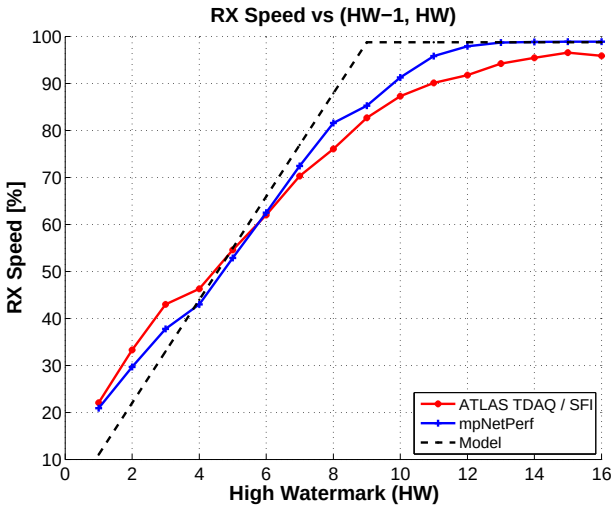


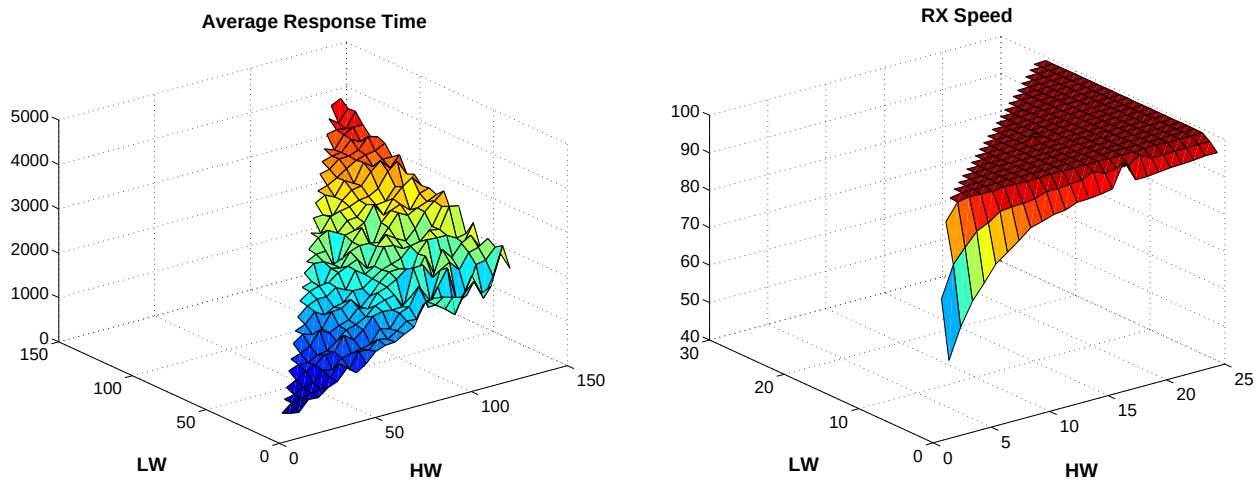
Figure 4.29: Comparison between mpNetPerf and the ATLAS software.

4.7.2 Queueing

The internal queues of *mpNetPerf* behave differently than those of GETB. In software, data has to traverse the operating system layers and it may have to wait in the kernel buffers before being delivered to the application. Our attempts in monitoring the internal queue in the server proved unsuccessful – even in the cases when the server was clearly under heavy load, its *internal queue* of requests was almost empty. The interpretation was that the requests were waiting inside the operating system. Therefore, we were not able to reproduce all the measurements from Section 4.6.

From the point of view of *network queueing* (which is more important), *mpNetPerf* behaves in the same way as the GETB hardware. The queue occupancy at the output queue of a switch, on the port connected to a client, grows with the average of the watermarks, as given by expression (4.16).

Figure 4.30(a) shows the average round-trip time for a test like the one presented in Section 4.6.3 (increasing the watermarks so that the switch output buffer fills up; see also Figure 4.22). The surface is planar and proportional to $\frac{L_W+H_W}{2}$, as for the GETB test. The round-trip time measured in *mpNetPerf* is equivalent to the one-way response delay measured by the GETB[48]. The *RX Speed* (Figure 4.30(b)) is reaching line speed as fast as in the GETB case.



(a) Response Time (RTT)

(b) *RX Speed*

Figure 4.30: mpNetPerf: Filling the output buffer of a switch.

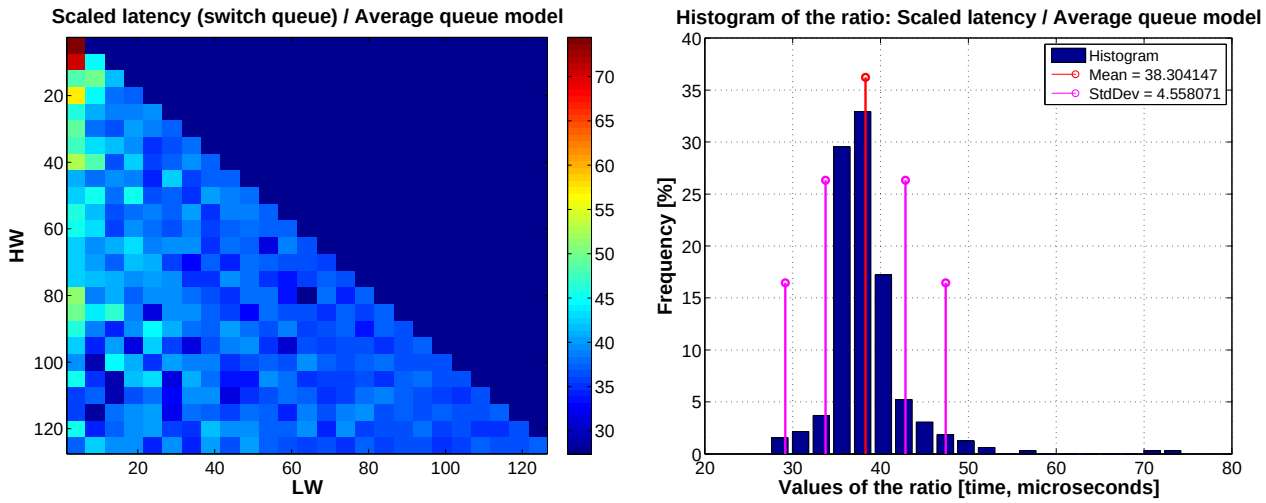
As we did in Section 4.6.3 and Figure 4.23, we tried to estimate the proportionality factor k between the response time and the expected surface $\frac{L_W+H_W}{2}$. Figure 4.31 shows the results. The "constant" k is now closer to the expected value $T_{rpl} = 36.9\mu s$. The ratio $100 \cdot \frac{\sigma}{\mu} = 11.8\%$ is bigger than in the GETB case.

4.7.2.1 Queue occupancy using the GETB

A more accurate way to do buffer occupancy measurements it to use the GETB tester and record the one-way latency, instead of the Round-Trip Time. Figure 4.32 shows the required setup for such a test.

The transmission and reception nodes are connected to the two switches SW_1 and SW_2. Traffic flows between the switches through the uplink L. While a client-server application runs between ports A, B (servers) and C (client), we can send packets between the two tester ports, T_1 and T_2. The probe traffic will traverse the same link L as the main application traffic. The probe packets will have to queue at the output of switch SW_1. At T_2 we measure their latency and infer the occupancy of the queue.

[48]The RTT includes the delay of the requests as a constant component and the delay of responses as a variable component. The latency for requests does not vary much because their network path is lightly used. The responses arrive sooner or later, depending on the network queue lengths.



(a) Image map of the ratio

(b) Histogram of the values

Figure 4.31: mpNetPerf: Proportionality factor: Latency in switch / Average queue model.

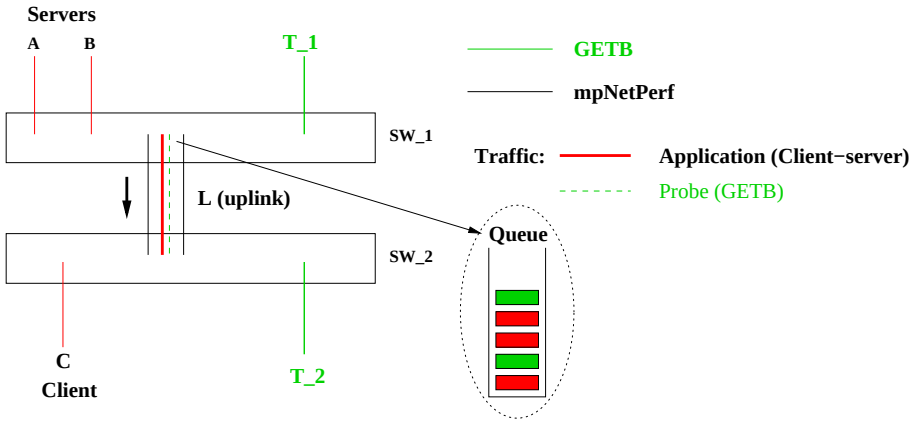
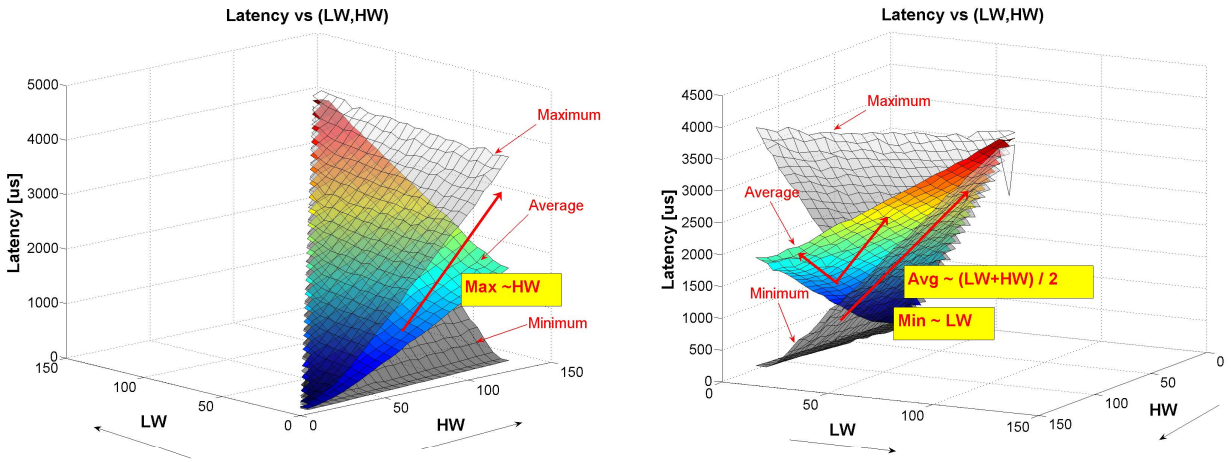


Figure 4.32: Measuring queue occupancy using the GETB.

We tried this setup using the *mpNetPerf* software running on ports A, B and C. After doing a *scan* of all the watermarks values in the range 125, we obtained the surfaces shown in Figure 4.33. For each combination (L_W, H_W) we saved the minimum, average and maximum latencies measured by the GETB. The resulting surfaces are depicted in Figure 4.33. They confirm the queue occupancy expressions from Section 4.6.2, i.e. $min \propto L_W$, $max \propto H_W$ and $average \propto \frac{L_W+H_W}{2}$.



(a) View 1

(b) View 2

Figure 4.33: Latency (queue occupancy) using the GETB (3D views).

4.7.3 Performance

The performance of *mpNetPerf* depends on the resources available on the host machine. As the program uses multiple threads (for reception and transmission of messages), it works better on computers with multiple processors. The computing power is more important for the client nodes as they involve more processing for the generation of requests and for keeping track of tokens.

At the client, the transmission and reception are linked by the token counter. This shared resource is the main bottleneck in the software implementation. The T_C is accessed at each reply received (or request sent). In order to fill its input line, the client has to send requests fast enough. The sending rate necessary for line speed decreases if the replies are larger. For certain reply sizes, the software simply cannot send requests fast enough to be able to maximize the *RX Speed*. Figure 4.34 shows the performance of an *mpNetPerf* client versus the size of the response. We see that on our test equipment, the client can reach line-speed only for replies larger than 4200 bytes. This corresponds to a maximum request sending rate of 28000 requests/second[49].

In the event of a client becoming CPU-limited, it will send requests slower and so its input rate will also drop. The slow-down effect remains localized on that particular client. When a

[49]The GETB does not have this limitation – the client can reach line speed for any reply size.

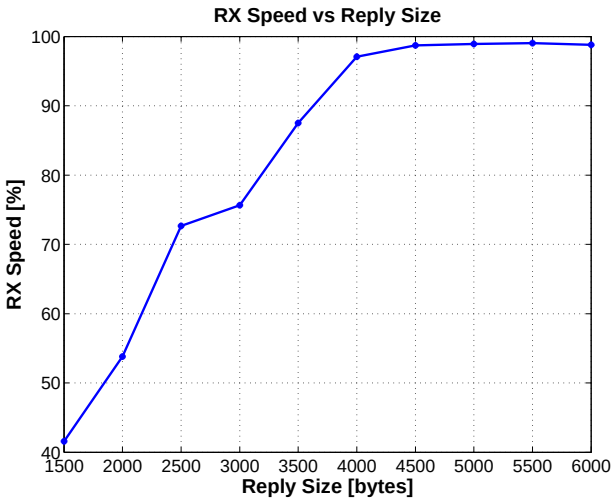


Figure 4.34: mpNetPerf: Client's performance versus reply size.

server becomes busy and cannot serve the requests fast enough, it affects the entire system. The busy server responds slower to requests, so *all* clients talking to it need more time to resolve the outstanding tokens (requests). The overall request sending rate decreases. Figure 4.35 shows a test we did with two servers and one client. After 27 seconds from the start of the test, one of the servers, which runs an *mpNetPerf* instance, is slowed down by another application which needs all the CPU time. During this test we monitored for each server: the transmission speed and the average time to answer (round trip time at the client). As there are two servers, each one should send at 50% to the client. We observe that even if the service time increases for one server, the transmission speed decreases for both of them – the fast server "follows" the slow one. As expected, the RTT increases only for the "slow" server. After another 23s the CPU becomes available again and transmission resumes at full speed.

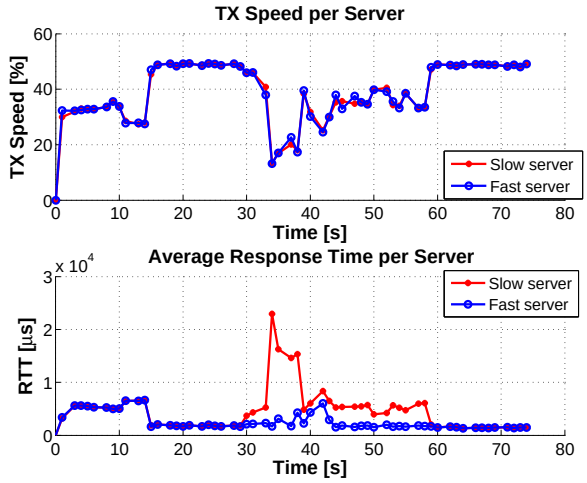


Figure 4.35: Effects of a server with a busy CPU.

4.8 Applications

The two tools we've developed, the GETB and *mpNetPerf*, can also be used to investigate certain features of switches. The fact that the watermarks determine the amount of data in the network allows us to measure the buffering capacity. Being able to detect when messages are lost gives us an indication when the buffers are full. And measuring the one-way latency or the round-trip time provides information about the degree of occupancy of the queues. The GETB hardware is useful for accurate measurements, while *mpNetPerf* has the advantage of easier deployment[50]. In addition, *mpNetPerf* can quantify the level of performance of a particular hardware/software platform.

4.8.1 Measuring buffer sizes and utilization

From Section 4.6.2 we've learned that the low and high watermarks determine the minimum and maximum queue occupancies. If $L_W = H_W - 1$ then the occupancy will be fixed at the value of H_W [51]. When multiple servers send data to a single client, the watermarks tell how much data is kept temporarily in the output buffer of the switch[52]. If we increase them and monitor the RTT (or the response delay), we'll notice a linear growth as the buffers are getting full, followed by a flattening RTT when the buffers fill up (at that point the client starts to lose messages). The size of the buffer can be computed as:

$$\text{Buffer Size} = \left(H_W^{\text{Max RTT}} - H_W^{100\%}\right) \cdot S_{rpl} \tag{4.17}$$

The buffer starts to fill only after the *RX Speed* at the client reaches line speed (for $H_W = H_W^{100\%} = N_C$) – this is why we substract this value when we compute the buffer size. We used this method to find the buffer depth for two switches, which had sizes of 1 Mbyte and 2 Mbyte. Figure 4.36 shows the plots of the RTT versus H_W. The size of a reply message was 10 Kbyte. We observe the flattening of the curves when the buffers are full, for $H_W = 100$ and $H_W = 200$ – the bending point of the curves gives the size of the buffer. For example: $H_W = 100 \rightarrow \text{Buffer} = 100 \cdot 10\text{Kbyte} = 1\text{Mbyte}$. This measurement was done using *mpNetPerf*.

If we know the RTT corresponding to a full buffer, it is also possible to estimate the current buffer utilization (even in real-time, on a "live" network, not on a test-bed). It is enough to launch a client-server application such that the client is at the output port where we want to estimate the occupancy. Then, issuing a few requests and measuring their response time, we get the buffer utilization (considering the linearity of the curves from Figure 4.36). This technique can be used to estimate the buffer occupancies during an ATLAS data-taking run[53]. Periodical

[50] Basically any computer able to run the ATLAS software can be used to start an *mpNetPerf* instance.

[51] When the network capacity is negligible, as for Ethernet switches.

[52] We discussed the buffering capacity of a switch port in Section 3.4.2 and the occupancy in Section 4.7.2.1. Those methods were based on the GETB hardware. In this section we present a method that can estimate the end-to-end effective buffering capacity or occupancy. It has the advantage that it is based on pure software so it can run between any two points of a network.

[53] This measurement method is *intrusive*. However, it can run in a fraction of a second, sending a few packets. The buffers in the network should be able to cope with a short increase of the amount of traffic.

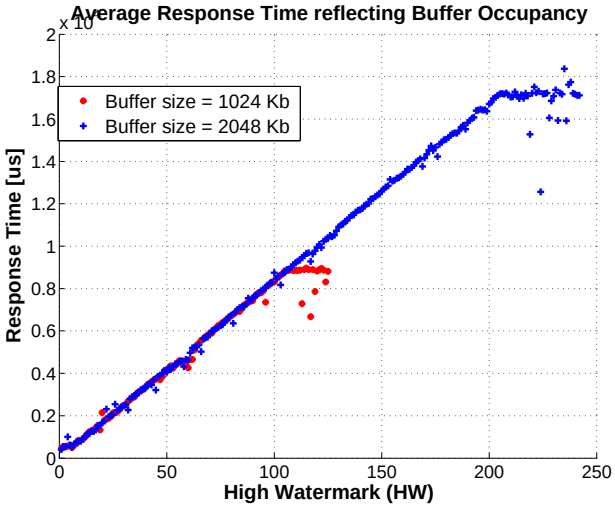


Figure 4.36: Buffer size and occupancy using the RTT / Response Time.

checks could be performed and alarms could be triggered if the occupancies grow beyond a certain threshold.

4.8.2 Assessing network performance

The ATLAS DAQ network has been designed to provide enough bandwidth for a system running at the maximum trigger rate. Because of its complexity, the components of the DAQ are developed and tested independently on smaller scales. When the entire chain is put into operation and it does not behave as expected, the diagnosis of a problem becomes a difficult task. A useful application of *mpNetPerf* is to test and validate the performance of the entire DAQ network. If the bandwidth measured by *mpNetPerf* is within the baseline[54] then a network issue can be excluded. *mpNetPerf* is based on the same communication libraries and uses the same type of request-response protocol so it reports the upper bounds in terms of transaction rates. The tunning of the watermarks is also easier using a lightweight application like *mpNetPerf*.

4.9 Remarks regarding the ATLAS traffic pattern

The aim of this chapter was to understand the interaction between the token-based traffic shaping mechanism and the underlying network. We are now able to make the following observations.

- The results we obtained cover the cases which appear in ATLAS (Section 4.2.2): the L2PU and the SFI clients and the ROS servers. We observe that the combination of parameters used by the SFI ($L_W = H_W - N$) has the most predictable performance (linear) and is

[54]The ATLAS Technical Design Report (TDR) [5] specifies the event rates and the bandwidth at which the system should run at different stages during its installation. That represents the baseline.

less "bursty" because it produces a constant stream of data. The L2PU uses low values for L_W; this leads to bursts of data from the ROSs and can also lead to temporary network congestion.

- We can answer now the "questions" from the end of Section 4.2.2 (page 78). The formulas we constructed allows us to predict the performance if the network delay is known and if the amount of buffering is also known (these two can be measured using the tools we developed). The "best" combination of watermarks is the one that offers top performance with the least amount of buffering, i.e. $H_W = N_C$ and $L_W = N_C - 1$. If we know the configuration of the TDAQ, we can say if the system will work on a certain network (measured in advance). Usually this reduces to checking if the watermarks are within the bounds of the network capacity and the maximum buffering capacity.

- Our analysis used as a starting point the description of a communication protocol. From there we employed a set of measurements and we derived analytical expressions to describe the bandwidth usage and queue occupancies. While this analysis applies to the particular case of the token-based traffic shaping, the methodology should be applicable to other flow control mechanisms as well. The GETB platform becomes valuable when we need to study the packet-level effects of the protocol specification.

- From the expression of the expected input rate at the client ((4.10) on page 89) we can observe that the Low Watermark parameter is closely related to the network capacity (N_C). Practically, L_W can eliminate the effects of the network delay.

- The other parameter, the High Watermark, determines the rate and if there will be buffering along the path from the servers to the clients. If H_W increases too much, there will be lost replies (see Section 4.6.2 on page 101).

- In this chapter we introduced the notion of the "network capacity" which is directly proportional to the network delay (round trip time) ((4.8) on page 87). If not properly taken into account (by adjusting the watermarks) the network delay will put an upper limit on the effective rate at the client. While the impact of N_C is small in a local network, it becomes significant for long-distance connections. ATLAS foresees to deploy parts of the TDAQ system on remote sites (outside CERN); for example, a Level 2 farm operating over a link with a long delay will have to be configured appropriately (by making $L_W > N_C$).

- The tools we've developed can measure the performance of a network, but they can also tell how much head-room for buffering is available. Normally, when the Low Watermark reaches N_C, the client receives data at line-speed. If the watermarks are higher than this limit, then the data is buffered either in the server, either in the network. The use of the GETB or *mpNetPerf* allows the determination of the excess buffering capacity. Sometimes it may be desirable to make the watermarks bigger than N_C just to assure a permanent 100% network utilization.

- In our analysis we considered that each client has a certain amount of buffering space allocated on the switch port where it is connected to. Therefore we assumed that clients connected to the same switch should not interfere one with the other. This may not always be the case. During the ATLAS switch evaluation program, we discovered using the GETB

Network Tester (Section 3.3) that some devices use *shared buffers*, i.e. they allocate a memory pool to a number of ports (usually 12). With such a device, the buffering space available to the clients decreases. Our tools allow us to discover if the switches have shared buffers; then we can take the appropriate actions. Usually, this means to reconfigure the switch firmware (and allocate individual buffer space for each port) or to connect the clients to the switch such that there is minimal overlapping of the (shared) buffers.

- When the shared buffers cannot be avoided then a modification to the traffic shaping mechanism might be necessary in order allow "horizontal communication" between the clients which share the buffer space. The clients should use a common "global" token counter and the watermark limits (L_W,H_W) should be enforced on this global variable. By using a global set of tokens, we can make sure we never overflow the buffers in the network. The main disadvantage of this kind of "horizontal" communication is that it relies on a distributed Inter-Process Communication protocol which has to be very efficient. The advantage is that it provides a dynamic allocation of the shared buffer space. Another alternative to cope with shared buffers is to statically configure each client to use only a fraction of the shared buffer space (B). Assuming there are N clients, then each client should use at most $\frac{B}{N}$ bytes from the buffer (this limit is imposed using the High Watermark: $H_W = \frac{B}{N \cdot S_{rpl}}$).
- Our last remark refers to the way the watermarks are configured. In this chapter we considered them as configuration constants. One can envisage a system where the watermark limits change dynamically as a function of the network state. For the data collection applications (SFI), when the clients detect congestion, they could decrease the watermarks and when they sense free network bandwidth, they could increase the watermarks. The congestion can be detected by monitoring the variations of the delay (or round trip time):
  - The client should keep track of the delays to/from all the servers it talks to. If one or more servers are slow to respond then we may have the situation described in Section 4.7.3. Usually, the client application cannot solve this problem by itself, but it may issue a warning message.
  - If the client notices increased delays from all the servers then this is an indication of queues filling up and of imminent network congestion. The watermarks should be decreased until the delays are stabilized.

4.10 Summary

In this chapter we studied the ATLAS request-response traffic and the associated token-based traffic shaping mechanism. For an ideal implementation of the protocol (i.e. hardware-based), we derived an analytic formula for the traffic rate in function of the token watermarks. The dependency between the utilization of queues and the watermarks has also been determined. We've shown that L_W, the low watermark, is related to the network delay, while the high watermark, H_W, is associated to the buffering capacity. Experimental results have been compared to the theory and a good agreement has been found. At the end we outlined practical applications of the hardware and software tools we've developed.

Part III

Installation and commissioning

Chapter 5

Tools for network management

The ATLAS network will contain hundreds of devices interconnecting thousands of nodes. During the installation and commissioning phases of the network, it became obvious that a network of this size cannot be maintained and supervised by human operators without using a Network Management System (NMS).

In this chapter we introduce the basic concepts in network management and then we focus on two components of a NMS, which are essential in the installation phase of a network, namely the configuration of devices and the discovery of the inter-connections between them (network topology). The performance monitoring aspect of a NMS will be covered in the next chapter.

5.1 What is a network management system

A NMS is a computer program that assists the network administrator in the typical tasks that are involved in running a computer network: configuration, health and performance monitoring, troubleshooting, etc. In the following we outline the requirements of a complete NMS. The X.700 recommendation from ISO [41] defines the following categories of functions that should be available in a management system:

Fault management Fault detection, isolation and correction. This includes functions for maintaining error logs, receive notifications and carry out diagnostic tests.

Accounting management This covers billing and charges for the use of the network. This category is oriented to service providers[1].

Configuration management This category includes the initialization of devices, setting up their operating parameters, tracking changes in their configurations, etc.

Performance management Functions for gathering statistical information are classified here. Measuring performance records (over time) and comparing them to base lines are also part of this category.

[1] Internet service providers, Telecommunication companies, Broadband carriers, etc.

Security management In this category we have all functions related to security and access control.

Systems that "do everything" with respect to the above categories are rare and quite expensive. On the market we can find both commercial and free products that can handle a part of the job of managing the network. We distinguish two broad categories of features: some which are useful to *install* the network and others which become more suitable when the network is put into *operation*. Below is a list of functional requirements, divided into these two categories.

Features required during the **Installation** phase – these are needed to ease the installation and then certify that the network is ready for production:

Device inventory The NMS should be able to create and maintain an inventory of the installed network equipment (at least) and of the end-nodes (desirable). For any device, this inventory should contain the location in the network and a reference to other sources (databases) having more information about the particular device.

Topology discovery The relationships between devices is as important as their proper registration in the inventory database. Network devices and their inter-connections (cabling) are installed according to well defined plans. After the installation is finished, it is good practice to cross-check the initial "drawings" (schematics) against what is in place. To do this automatically, the physical connections have to be known, i.e. the topology of the network. After the deployment (and initial verification), the topology should be constantly monitored for unauthorized changes[2]. In addition to its use for documenting and validating the network installation, knowledge of the topology is useful for troubleshooting malfunctions.

Device configuration Each device in the network (switch or router) needs to be configured to inter-operate with its peers. This task becomes complicated and error-prone when there are many devices, and especially when they are not identical[3]. A system able to automatically setup the devices (in the installation phase) and then track the configuration changes (in the operation phase) is essential for keeping the network under control.

Diagnostics Before being put into operation, the network should be tested – both for connectivity and performance. Such tests are obligatory for commissioning, but they can also be used during the operation phase, to debug network issues. A complete NM system should include diagnostic tools.

Features required during the **Operation** phase – these features are oriented towards monitoring the state of health and the performance of a running system:

[2]Example of changes that should be detected: cables unplugged or moved by mistake, new devices connected without proper registration, etc.

[3]In ATLAS, we shall use network equipment from a few different manufacturers. We shall also have to support different revisions from the same line of products.

Health monitoring – Listening to alarms and notifications Most network devices implement the Simple Network Management Protocol (SNMP) [42] for monitoring and for performing simple configuration tasks. SNMP has a mechanism which allows the device to send notifications to another entity. These are called *SNMP traps* and can be generated from a variety of causes: link failures, configuration changes, state-machine changes inside protocols, etc. Some of these traps may be caused by real problems, others just happen during normal operation[4]. The NMS should be able to receive all these messages, filter them if necessary and use them in case something goes wrong (to perform the Root Cause Analysis, described next).

Root Cause Analysis (RCA) This is a method aimed in identifying the *real cause* of a problem. By analyzing the symptoms, it tries to identify the most likely source that produced it. RCA works by correlating failure events with knowledge about the hardware, software, the current state and the configuration of the failing system. The goal is to pinpoint and possibly fix the real problem, minimizing the downtime. Artificial intelligence and the theory of expert systems are often employed for implementing RCA engines.

Integration with the ATLAS control infrastructure It is often useful to integrate the NMS with other 3^{rd} party applications. The ATLAS control infrastructure (the Online Software) is used by physicists to configure and monitor the status of the data-taking in the experiment. The state of the TDAQ depends on the health of the network. Therefore, it is expected that the NMS is able to provide information to the Online Software. This can be done if the NMS has an Application Programming Interface (API) that can be used to extract data from its knowledge base.

Traffic/performance monitoring The TDAQ network has been designed and dimensioned to sustain the expected traffic rates. However, during the operation phase, it can easily happen that this rates are exceeded. The NMS has to monitor the traffic levels on all the links in the network and generate alarms if these grow over some predefined thresholds. The lack of bandwidth (congestion) during data-taking is a serious issue. When congestion happens, the NMS should be able to quickly find the hot spots, identify the causes and notify the Online Software infrastructure. This, in turn, could simply inform the operator or trigger more advanced actions in order to alleviate the congestion.

Visualization The task of visualizing the state of a big network, as that of the TDAQ, comprising thousands of devices, is a real challenge. In such cases, a hierarchical approach has to be used, where status and statistics are aggregated to provide more or less detailed views of what is happening in the network. Visualizing large networks is still a subject of active research.

After a market survey, ATLAS has settled on a commercial product[5] that fulfills many (but not all) of these requirements [43]. This product will be complemented with custom developed tools. The advantage of custom designs is that they can be tailored to match very well the

[4]For example, an SNMP trap which announces that a link is down might explain why a certain part of the network is not accessible. On the other hand, an SNMP trap notifying that BGP (Border Gateway Routing Protocol) has found a new neighbor might be safely ignored.

[5]Spectrum, produced by Aprisma / Computer Associates.

specific needs and they can allow for easy integration with the rest of the infrastructure. Their main disadvantage is the lack of long-term support. As the ATLAS experiment is foreseen to run for 10 years or more, the future maintenance of the network and its management system is an important factor to take into account. The operational model of the ATLAS network has been described in [44] and, more recently, in [45].

In this chapter and the next one we shall concentrate on a few particular aspects of network management. We shall describe solutions that were used during the installation phase (in the current chapter) and our plans for the operation phase (in Chapter 6). In Section 5.2 we present the tools we used to configure the devices. Then in Section 5.3 we discuss the problem of discovering the physical topology of an Ethernet network; our solution will be presented as well.

5.2 Device configuration

Any modern switch (or router) has a command-line interface (CLI) used for configuration. This interface can be accessed via the serial console or by connecting using a remote access program[6] to the device. The CLI syntax and the available commands are specific to a family of devices (from the same vendor). There is no standardized command-line set that works for devices from all manufacturers.

In addition to the command line interface, most devices contain also SNMP agents. The primary function of SNMP is to read device status and statistics. Only a limited set of parameters can actually be *set* using SNMP. The bulk of the configuration process has to be done using the command line[7].

The network industry has realized the drawbacks of having proprietary configuration methods for each manufacturer – the IETF[8] has devised in 2006 the Netconf standard [46]. Netconf provides a mechanism to describe and deploy configurations using an XML syntax[9]. Unfortunately, at the time of this writing there were no vendors implementing Netconf. However, many manufacturers expressed interest in this standard – we expect it will gain a wider acceptance in the next years. Until then, the command line remains the only way to set up the devices.

5.2.1 The problem

The use of the CLI to manually enter commands is not a method that scales to more than a few switches. When we have to deal with tens of (possibly different) devices, the job of setting their

[6] *telnet*, SSH (Secure Shell) or any serial terminal emulator like *minicom* or *Hyper Terminal*.

[7] In addition to the CLI and to SNMP, some devices can download their settings from a file server. Such a configuration file is a list of commands that are interpreted by the device, as if they were typed on the command line. In general, this feature is available only at startup, when the device is powered on.

[8] Internet Engineering Task Force, an organization that develops and promotes Internet standards.

[9] The Extensible Markup Language (XML) is a general-purpose markup language. Its primary purpose is to facilitate the sharing of data across different information systems.

parameters becomes quite tedious. Even a minor change would take a long time to complete by hand and would be susceptible to errors.

It is possible to run scripts based on the remote access programs. A major inconvenience of these scripts is that they run blindly: starting from a known configuration, they can send a sequence of commands assuming they are executed correctly by the device. If something goes wrong at one point, all the subsequent commands will fail as well. If the user wants to embed some "intelligence" in these scripts – to implement, for example, error recovery and check the results of the execution – then a more powerful programming language is required.

5.2.2 A solution

We developed a tool to ease the job of applying configuration changes in the ATLAS network. We called it *sw_script* because it allows us to write scripts (short programs) for managing switches or routers. *Sw_script* is a Python module which provides an abstract, object-oriented, way of interaction to network devices. Internally, the module uses *telnet* and SNMP for communication with the managed devices. It can issue commands and knows how to interpret the response of the target. In some cases it *normalizes* the output, i.e. no matter what type of switch, *sw_script* will return the results in the same format (an example is given in Listing 5.2.2).

The module offers a generic function interface which covers the most common configuration and status monitoring tasks. When using *sw_script*, the developer creates a Python object and calls its member functions to control the switch. For a different device, the user has just to instantiate a different kind of object (another "class"). The implementation of the access functions may differ from one device to another, but these details are transparent to the user. Basically, *sw_script* brings all flavors of command line interfaces to a common denominator[10]. Listing 5.2.2 contains a sample interactive Python session with *sw_script* (more examples are available in [47]).

The *sw_script* module has been used in several projects:

- The GETB tester uses *sw_script* in order to change the configuration of the DUT[11]. In this way, we can run the same tests in multiple configurations, without any manual intervention from the user. After each test iteration, the tester uses *sw_script* to retrieve the statistics from the DUT. These are compared to the counters from the GETB and any mismatch is promptly reported.

- For the TDAQ network, we developed several programs using *sw_script*– they have been successfully used to configure tens of devices in the network. Firmware upgrades[12] on all devices were also made possible using *sw_script*.

- We also created monitoring tools using this module as the low-level device communication interface. For example, we created a web-based interface that allows the user to "browse"

[10] *sw_script* currently supports 13 device models from 7 different manufacturers.
[11] Device Under Test.
[12] A *firmware upgrade* is the procedure by which the operating system of a device is changed. The device can be a switch or a router.

Listing 5.2.2 – An interactive session with *sw_script*.

```
# Load the sw_script module
>>> import sw_script

# Create an object associated to the switch (a Cisco device in this case)
>>> sw = sw_script.Cisco_Catalyst_6500_Switch(ip_address = "10.2.197.240", password = "cisco");

# List the ports available on this device
>>> sw.get_port_names();
   ['1/1', '1/2', '1/3', '1/4',  ....

# Get all the information available for an interface
>>> sw.get("2/4");
                [('rx_packets', 519.0),  ('rx_bytes', 127937.0),  ('rx_multicast', 1034.0),
                       ('rx_discards', 0.0), ('rx_errors', 0.0),
                 ('tx_packets', 11199.0),('tx_bytes', 1111661.0), ('tx_multicast', 5011.0),
                        ('tx_discards', 0.0), ('tx_errors', 0.0),
                 ('description', 'GigabitEthernet2/4'),
                 ('link_state', 'up'),
                 ('mac_addr', ['00:90:27:8F:94:E3'])]
# This will return the counters (number of in/out packets) and
# also the associated MAC address table, i.e. a list of devices
# that are accessible via this interface.

# Execute a command directly on the switch (as if it was typed by the operator)
# We print the output of the command (as it would appear on the operator console)
>>> print sw.exec_cmd("show mac-address-table")
  Unicast Entries
   vlan   mac address     type        protocols               port
  -------+---------------+--------+---------------------+--------------------
   1    000d.ed90.907f    static ip,ipx,assigned,other  Switch
   1    00eb.5900.0700   dynamic other                  GigabitEthernet2/5
   1    00eb.5900.0701   dynamic other                  GigabitEthernet3/1
   1    00eb.5900.0800   dynamic other                  GigabitEthernet4/6
   1    00eb.5900.0801   dynamic other                  GigabitEthernet5/4
   1    00eb.5900.0900   dynamic other                  GigabitEthernet3/6

# The command above shows all the MAC addresses the switch knows about.
# Each switch has its own way of displaying this table.
# We use now a sw_script function to retrieve the same information.
>>> sw.mac_address_table()
      [('00:EB:59:00:07:00', '2/5'),
       ('00:EB:59:00:07:01', '3/1'),
       ('00:EB:59:00:08:00', '4/6'),
       ('00:EB:59:00:08:01', '5/4'),
       ('00:EB:59:00:09:00', '3/6')]
# The mac_address_table() function returns a simple Python list with the addresses
# and the port names. A similar result would be obtained on any device supported by sw_script.
```

through the installed network, observing the status of the switches, viewing traffic plots for each port and searching for particular end-nodes.

- A network discovery tool has been created using *sw_script*. Starting from a list of devices (switches), this program scans the entire network and generates an inventory report with details about all devices and their inter-connections. The method used for discovery is described in the next section.

5.3 Topology discovery

Maintaining an accurate and complete knowledge database of the physical network topology is a prerequisite to many critical network management tasks, including network diagnostics, resource management, event correlation, root cause analysis and server placement.

This knowledge refers to the actual physical connections between the existing network elements. Due to the frequent changes of the element connectivity, accurate topology information, cannot be practically maintained without the aid of automatic topology discovery tools. In spite of its critical role for network management, it is not always straightforward to obtain this information. There are two classes of topological maps that can be constructed for a network:

Network layer map (Layer-3) This refers to the map as it is seen by the IP routing layer[13]. It can be regarded as a logical map of the network, containing only the connections between routers. Such a "connection" can sometimes hide an entire infrastructure built with Layer-2 or Layer-1 devices.

Data link layer map (Layer-2) This reflects more closely the physical layout of the network, i.e. the real connections between switches and routers and between computers and switches[14]. Data link maps are essential for the management of the network. They complement the Layer-3 maps.

The first type of map, the one containing routers, is quite easy to obtain since routers are aware of their immediate Layer-3 neighbors as well as the attached subnets. They make this information accessible via SNMP, which is sufficient to determine the Layer-3 topology. But this fails to capture the complex interconnections of the Ethernet LANs, that underlie the logical links of Layer-3. Unfortunately, Layer-2 elements (switches), do not provide similar information about other devices in their immediate vicinity. This complicates the discovery of the physical network topology.

The need for an up-to-date Layer-2 topology information has been recognized by both the telecommunication industry and the research community. Many manufacturers have already

[13]The network or IP layer is the 3^{rd} layer in the OSI stack. Layer-2 refers to the Ethernet switching layer (data link). See Appendix A for details.

[14]There are cases when the Layer-1 map (the physical medium layer) is not entirely reflected in the Layer-2 description. In general we can say that a Layer-N map will not contain devices which work only at Layer-$(N-1)$. Repeaters, for example, will not be discovered by a Layer-2 mapping. Modern Ethernet networks do not employ Layer-1 devices very often, therefore in this work we assume the Layer-2 and Layer-1 maps are identical.

deployed their own proprietary protocols for discovering physical interconnections. Examples include the Cisco Discovery Protocol (CDP) and the Extreme Discovery Protocol (EDP). These are proprietary protocols and are not applicable in a heterogeneous, multi-vendor environment[15].

In this section we shall discuss how to determine the physical topology of a local Ethernet network. We begin by explaining the problem we are trying to solve and the data at our disposal (Section 5.3.1). In Section 5.3.2 we prove an important statement that will be the basis of the solution, while in Section 5.3.3 we use this statement to develop a simple algorithm which is able to infer the topology up to Layer-3. Finally, an implementation and sample results are presented in Section 5.3.5.

We do not attempt to study the topology discovery in its full complexity – we shall present only the issues relevant to plain Layer-2/3 networks like the TDAQ. The interested reader can refer to the works of Breitbart [48], Lowekamp [49] and Bejerano [50] for a more detailed presentation of the subject.

5.3.1 Statement of the problem

In the following we assume the reader is familiar with the way Ethernet switches work (if necessary, Appendix A.6 is available). We summarize below the basic concepts that govern Layer-2 switching:

- An Ethernet network is built as a tree of interconnected switches. Normally, there exists a single path between any two end-nodes – loops are illegal in Ethernet.
- Each device (switch or computer) that connects to Ethernet has a unique identity, its MAC address[16].
- Switches forward frames (packets) by using the information contained in their MAC address tables. These tables are constructed during a learning process, from the frames that pass through the device. Each physical interface has an associated entry in the MAC table. The entry for port P stores the addresses of the devices that can accessed via that port.
- When the switch receives a frame, it learns the source MAC address and then performs a lookup for the destination address. If the destination is found in the MAC table, the frame it forwarded to the corresponding interface. Otherwise, the frame is *flooded* to all interfaces.

[15]There exists an initiative called *IEEE 802.1ab* which proposes the *Link Layer Discovery Protocol* for Layer-2 neighbor discovery. It provides a method for switches, routers and wireless access points to advertise their identification, configuration and capabilities to neighboring devices. This allows a network management system to model the topology of the network by interrogating the SNMP databases in the devices. Unfortunately, this is not yet implemented by all manufacturers. And even if it would be, discovering the topology of networks containing legacy devices (which cannot be upgraded to support new protocols) still remains an open issue.

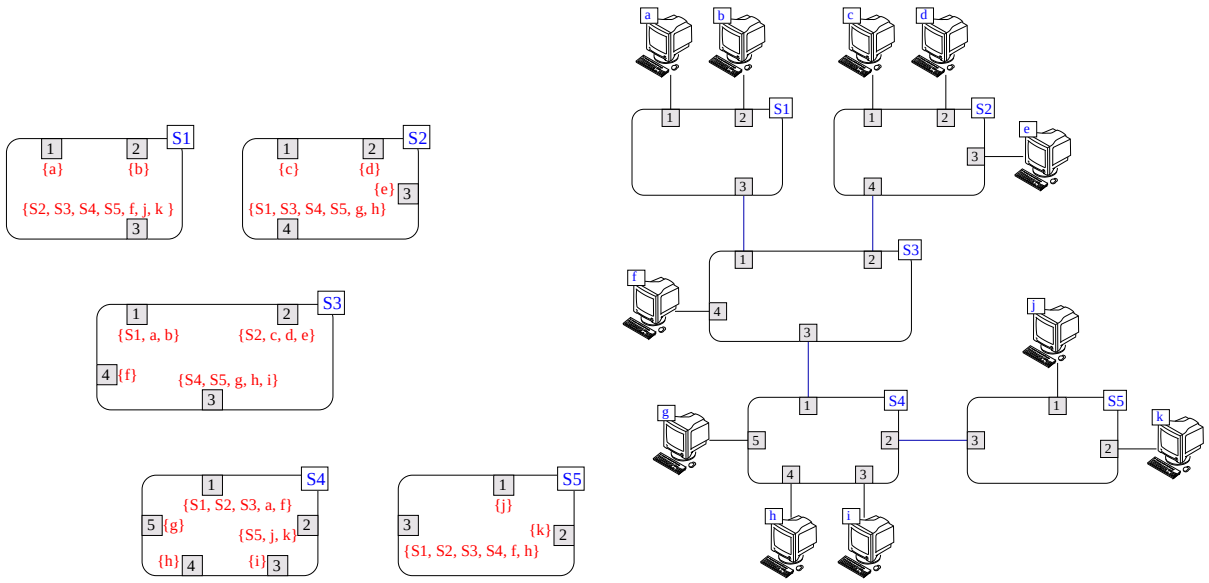
[16]MAC = Media Access Control. The MAC address is represented by a 48-bit number uniquely assigned by the device manufacturer.

- VLANs (Virtual LANs) can be used to partition an Ethernet network. Within a VLAN, loops are not allowed. On a given switch, each VLAN has its own MAC table. A network that contains VLANs is equivalent to multiple independent networks.

The Ethernet standard does not offer the possibility of interrogating the switches about their neighbors. One way to discover the close neighbors is to use the local knowledge from each switch (i.e. its MAC address table) in order to find the inter-switch physical interconnections (the *uplinks*).

By gathering the MAC tables from all the switches in the network and by correlating the information between them, it is possible to determine the position and the end points of all uplinks. The network location of the end-nodes can be also extracted from the MAC tables[17].

In Figure 5.1 we show an Ethernet network with five switches. The figure contains also the MAC addresses that were learned on the ports of each switch. On side (a) we show the input data we can use: the MAC tables from all switches (the uplinks were hidden because these are the unknowns). On side (b) we show what we expect to obtain from the discovery – a complete picture of the network with all the inter-connections. In these figures, lower case letters correspond to MAC addresses of computers, while uppercase Sn denote addresses of switches.



(a) Available data

(b) Expected result

Figure 5.1: The topology discovery problem.

In order to find a solution, but also to make the presentation easier to understand, we shall make the following simplifying assumptions:

[17]The switch interfaces that have only one address learned are most likely connected to a computer, not to another switch.

- There exists a method to obtain the MAC address table from any switch in the network[18].
- Any switch has its own MAC address and this address is known to all the other switches in the network[19].

Given these assumptions which hold for the ATLAS TDAQ network in particular and for enterprise networks in general, we present in the next section the result which will lead us to a method of finding the topology.

5.3.2 Direct Connection Theorem

The following statement specifies what conditions are necessary and sufficient for two switches to be directly connected by an uplink. This statement is called in the literature the *Direct Connection Theorem* [49].

Assume we have two switches S_1 and S_2 and we want to verify if they are linked by their ports A and B respectively. The MAC addresses of the switches are S_1 and S_2. To avoid confusion between switches and ports we shall write S/P to refer to the port P of switch S.

We denote by $U(S, P)$ the set of MAC addresses which are known by switch S via its port P. For example if one end-node with MAC address m_1 is connected to port P then we shall have $U(S, P) = \{m_1\}$. If the port P is connected to another switch Q having two hosts m_1, m_2 connected to it then we may have $U(S, P) = \{Q, m_1, m_2\}$. Sometimes we shall say that address X is *seen* by port S/P if $X \in U(S, P)$. A port P is an *uplink* if $|U(S, P)| > 1$.

If the assumptions from Section 5.3.1 hold, then:

There exists a direct link between the port S_1/A and the port S_2/B if and only if:

1. $S_2 \in U(S_1, A)$ – Switch S_2 can be reached via port S_1/A.
2. $S_1 \in U(S_2, B)$ – Switch S_1 can be reached via port S_2/B
3. $U(S_1, A) \cap U(S_2, B) = \emptyset$ – There are no addresses which can be accessed in the same time from both ports, S_1/A and S_2/B.

Proof:

$\Rightarrow$ *If ports S_1/A and S_2/B are connected then the three conditions hold.*

[18] In practice this is always true for the so-called *managed switches* – those which have a built-in CPU and support a long list of configurable features; these are also called *enterprise switches*. The assumption does not hold for *unmanaged switches* – devices which can only forward packets without supporting any advanced protocols; these are commonly installed in home networks and in offices.

[19] Again this is valid only for *managed devices*. For the self MAC address to be learned by the neighbors, it is necessary to have a protocol like Spanning Tree (STP) which forces periodic data exchanges between devices. This assumption if generally valid for any enterprise network.

We shall use the fact that in Ethernet there can be at most one path between any two nodes. As a consequence, if the two switches are directly connected, it means that all communication between nodes on the two "sides" has to pass over the link between them. This includes any communication involving the switches themselves (for example *pinging*[20] the switch IP address). Hence $S_2 \in U(S_1, A)$ and $S_1 \in U(S_2, B)$ so (1) and (2) are satisfied.

Let's assume now that there exists a direct link between S_1/A and S_2/B, but (3) is not true, i.e. there are nodes which are seen by both ports A and B: $U(S_1, A) \cap U(S_2, B) = \{k_1, k_2, ...k_n\}$. This implies that if a node connected to S_1 sends a frame destined to node k_1 then S_1 will see in its MAC table that it has to forward the frame to port A, towards switch S_2. Once the switch S_2 gets the frame, it will send it back to S_1 because it knows that $k_1 \in U(S_2, B)$. The original frame will go into a loop, which is illegal, therefore the assumption from the beginning is false and we cannot have common elements in the two sets: $U(S_1, A) \cap U(S_2, B) = \emptyset$.

$\Leftarrow$ *If all three assumptions are true then the ports A and B are directly connected.*

Let's assume that $U(S_1, A) \cap U(S_2, B) = \emptyset$, but S_1 and S_2 are not connected. So there has to be a path P between the two switches[21]. The path P is defined by the ordered set of MAC addresses and port numbers of all the devices that it intersects.

Therefore $P = \{S_1, T_1/P_1, T_2/P_2, ..., T_n/P_n, S_2\}$, where T_k/P_k are switches and ports along the path. Note that we did not yet say via which ports the path P enters the two switches S_1 and S_2. These ports may or may not be A and B, as we discuss below.

If path P contains ports S_1/A and S_2/B then it means that there is at least one other switch in the middle (T_k), between S_1/A and S_2/B (because we assume now that S_1 and S_2 are not directly connected). The MAC address of this switch should be "visible" from the ports A and B[22]. So we should have that $T_k \in U(S_1, A) \cap U(S_2, B)$, in contradiction to condition (3).

Assume, as the second possibility, that path P contains exactly one of S_1/A or S_2/B, or neither of the two. We know also that S_1 is reachable via S_2/B and S_2 via S_1/A[23]. So we reached a situation in which the switches are reachable via ports A and B on one hand, and via path P which does not contain *both* of these ports. Therefore we must have a loop in the network, which is again a contradiction. As we've shown that both implications are true, the statement is completely proven.

We can use the data from Figure 5.1 to check the connectivity between the switches, using the Direct Connection Theorem (DCT). Consider switches S_1 and S_2. First we find if S_1 sees the other switch and we see that yes, the switch S_2 is seen by port $S_1/3$. Switch S_2 also sees S_1 on port $S_2/4$. So the first two conditions are satisfied. But we observe that $U(S_1, 3) \cap U(S_2, 4) = \{S_3, S_4, S_5\}$, therefore these two switches cannot be connected. We now look at S_1 and S_3. We identify ports $S_1/3$ and $S_3/1$ as candidates. As the intersection of the MAC tables for these two ports is empty, we conclude there has to be an uplink between them.

[20] *ping* is a program that uses the IP/ICMP protocol to send *echo request* packets and expects to receive back *echo replies*. It is used to test IP connectivity. It can tell if a host is reachable in an IP network.

[21] Any two nodes in the same Ethernet network should be connected by one path.

[22] One of the assumptions in Section 5.3.1 is that the switches learn the MACs of *all* the other devices in the network.

[23] This is a result of assumptions (1) and (2): $S_1 \in U(S_2, B)$ and $S_2 \in U(S_1, A)$.

5.3.3 Algorithm description

The statement demonstrated in the previous section can be used to check the presence of a direct connection between any two switches, if all the three conditions are true. The algorithm proceeds first by collecting the MAC tables from all the switches. Then it filters these tables and keeps only the uplink ports. Finally it iterates over all pairs of switches and checks if DCT applies. The main procedure which scans the network and finds the connections between switches is presented in Algorithm 1. The function that does the connectivity test is shown in Algorithm 2.

Algorithm 1 Topology Discovery Algorithm

Phase 1
Gather the MAC address tables from all the switches in the network. Create a data structure called *switches*.
for S in *switches* **do**
  Filter the MAC table of switch S so that only uplinks remain.
  For each uplink of S remove the MACs of the end-nodes and keep
  only those of the other switches.
end for
Phase 2
Initialize G, the network graph.
for S_1 in *switches* **do**
  for S_2 in *switches* **do**
    $(A, B) \leftarrow$ test_for_direct_connection(S_1, S_2)
    if A and B do exist **then**
      Add the edge $(S_1/A, S_2/B)$ to the network graph G
    end if
  end for
end for

Algorithm 2 Test for Direct Connection

Input parameters: S_1, S_2
$A \leftarrow$ port on switch S_1 where the address of switch S_2 appears
$B \leftarrow$ port on switch S_2 where the address of switch S_1 appears
if both A and B exist **then**
  $U(S_1, A) \leftarrow$ set of nodes (addresses) known by port S_1/A
  $U(S_2, B) \leftarrow$ set of nodes (addresses) known by port S_2/B
  if $U(S_1, A) \cap U(S_2, B) = \emptyset$ **then**
    return (A, B)
  end if
end if

In all other cases return $Null$

This topology discovery algorithm works only if the switches in the network communicate between them using maintenance protocols like Spanning Tree or discovery protocols like CDP. In addition, it assumes all switches are "managed", enterprise-class devices. "Unmanaged" switches can be handled only if they are installed at the edges of the network (leaves in the network tree) – only then the algorithm can be adapted to infer their presence.

Another observation is that the algorithm disregards completely the information about the end-nodes and requires that the MAC addresses of the switches to be known everywhere in the network. The method described in [49] is more robust because it makes use of the complete MAC tables (with addresses of switches *and* of end-nodes). That approach allows the algorithm to tolerate more easily incomplete MAC tables.

5.3.4 Layer-3 extensions

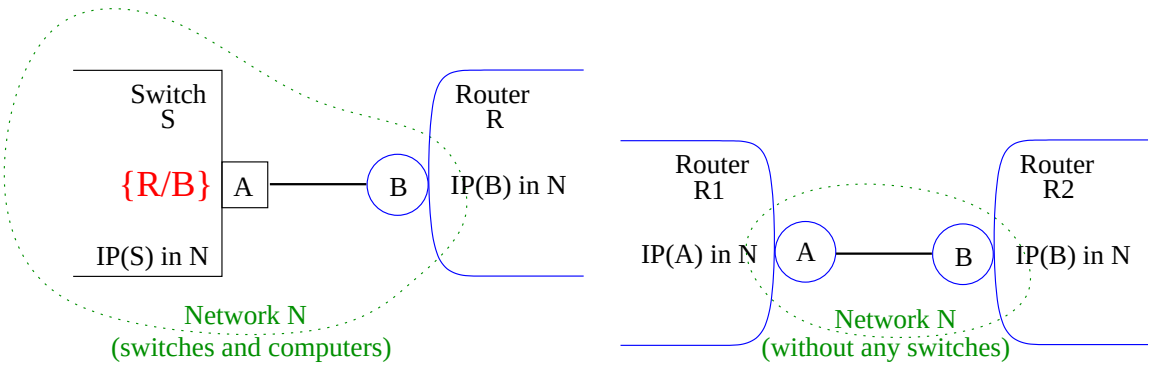
The method from Section 5.3.3 can infer the physical Layer-2 topology, but it will stop at router boundaries. The reason is that routers do not learn MAC addresses as they work only at the IP level. We describe below how can we extend the procedure such that it is able to discover connections between routers and between routers and switches.

Switch to router connections

We assume we have to deal with a local switched network N that contains hosts and switches in the same IP subnet. Such a local network will connect to the "outside" via a switch S/port A connected to a router R/interface B (Figure 5.2(a)). The interface on the router R must have an IP address belonging to the same subnet N. We assume there exists a physical connection between S/A and R/B if:

- $U(S, A) = \{R/B\}$, i.e. the MAC table on the switch interface contains *only one* address, the one of the router interface R/B[24].
- $\{IP(R/B), IP(S)\} \subset N$, i.e. all IP addresses (router interface and switch) are in subnet N.

If the switch S knows *only* about the MAC address of the router on port S/A then it means that there is a Layer-1 path up to that interface (otherwise the MAC table would contain more entries). It can happen that this path has some other unmanaged devices – these will be left undetected.



(a) Switch to Router

(b) Router to Router

Figure 5.2: Discovery of Layer-3 connections.

[24] A router has a unique MAC address associated to each interface.

Router to router connections

Direct connections between two routers are more difficult to trace. Assuming that we can discover and map all the switches in the network, we shall say that router interface R_1/A is directly connected to interface R_2/B if (see Figure 5.2(b)):

- $\{IP(R_1/A), IP(R_2/B)\} \in$ (same subnet) – The two router interfaces must be in the same IP subnet.
- No switches were found that were connected to R_1/A or to R_2/B (according to the rules from the previous section).

If these two requirements are met, we state that the two routers are directly connected. The main observation is that we assume that all Layer-2 (switch-to-switch) and Layer-2/3 (switch-to-router) connections have been found before we start looking for pure Layer-3, inter-router links.

5.3.5 Implementation

We implemented the main Layer-2 algorithm and its Layer-3 extensions in Python using the *sw_script* interface (Section 5.2.2). *Sw_script* is used to retrieve the MAC tables from switches and IP routing tables from routers.

For switches, in order to populate the MAC tables, our implementation makes use of broadcast *ping* messages[25]. This provides a simple way to make the switches learn all the addresses available. After this phase, the MAC tables have to be gathered quickly so that the entries that were just learned are not removed because of the aging[26].

To include unmanaged switches in the network graph, we infer their presence. If on one end of a link we see more than one address learned, and we have no information about the device at the other end, we assert that there has to be at least one switching device connected to that port.

The algorithm works in the presence of VLANs because switches maintain MAC tables on a per-VLAN basis. An additional requirement is that the loop-avoidance protocols (Spanning Tree) are enabled for each VLAN separately. This insures that the addresses of the switches themselves are known in all the VLANs.

The output of the topology discovery algorithm is a non-directed graph. The visualization of such a structure is not a trivial task. Graphviz [51] is a system that can produce bi-dimensional

[25]Normally, *ping* sends packets to a single host. A *broadcast ping* will issue a packet that will be replicated in the entire network and will be received by all hosts in the same IP subnet. With one packet we can check an entire range of hosts.

[26]A switch removes a MAC address from the table after a few minutes, if no frames are received from that source.

views of a graph if its textual description is given. We used this tool to draw our networks – it provides acceptable results, unfortunately not always predictable[27].

In Figure 5.4 (page 138) we show a view of the ATLAS network in its state at the time of this writing[28]. Three networks are shown: one for control and two for data traffic (FrontEnd and BackEnd). This installation contains almost 60 devices providing connectivity to over 500 computers. Most of the computers have two or three network cards – the total number of connections is close to 1000. The time required to collect the data and generate the graph is less than one minute. The loops that can be observed are due to the presence of VLANs. The star-points surrounded by many switches are the central (core) devices. They function as routers[29]. In the middle of the map there is a router that connects to the CERN network backbone. The discovery stops there because of access restrictions.

5.3.6 Applications

The discovery algorithm is now part of a complete network inventory application. This application scans the network and generates a report with all the devices and their inter-connections. By comparing two reports we can track the changes to the network. In the near future we plan to cross-check the output of the automatic discovery to the information from the other equipment databases used in ATLAS.

The network map generated by the discovery program has also been used for visualization. We implemented a traffic monitoring application using the *sw_script* module and a graph visualization application [52]. This application displays a 2D graph of the entire network (Figure 5.3). The color of each connection is changed in real-time as a function of its usage. This system allows us to see immediately which are the most used network segments. As color differences can be easily observed, we can also find out if the system is "balanced", i.e. if the links are equally loaded.

5.3.7 Other methods

We presented a method that can find the Layer-2/3 topology of an Ethernet network. It is based on the analysis of the MAC tables obtained from the switches. Other methods do exist, which do not require access to these tables (sometimes the access is restricted due to security policies).

The patents [53] and [54] describe an approach based on correlations between traffic patterns observed at different ports of the network. Their method infers network connections for ports with similar traffic characteristics.

In [55] researchers from Microsoft present a method based on a distributed system, which

[27] By this we mean that minor changes to the network structure can produce a completely different network graph layout. This happens because the layout algorithm uses a random starting point.

[28] The architecture of this network has been described in Section 2.4 (page 28).

[29] The devices that have been acquired for ATLAS can behave either as switches or routers. One can define interfaces which work at Layer-3 only, i.e. they forward packets based in IP addresses. In the same time, the device can have other interfaces with just plain Layer-2 switching functionality.

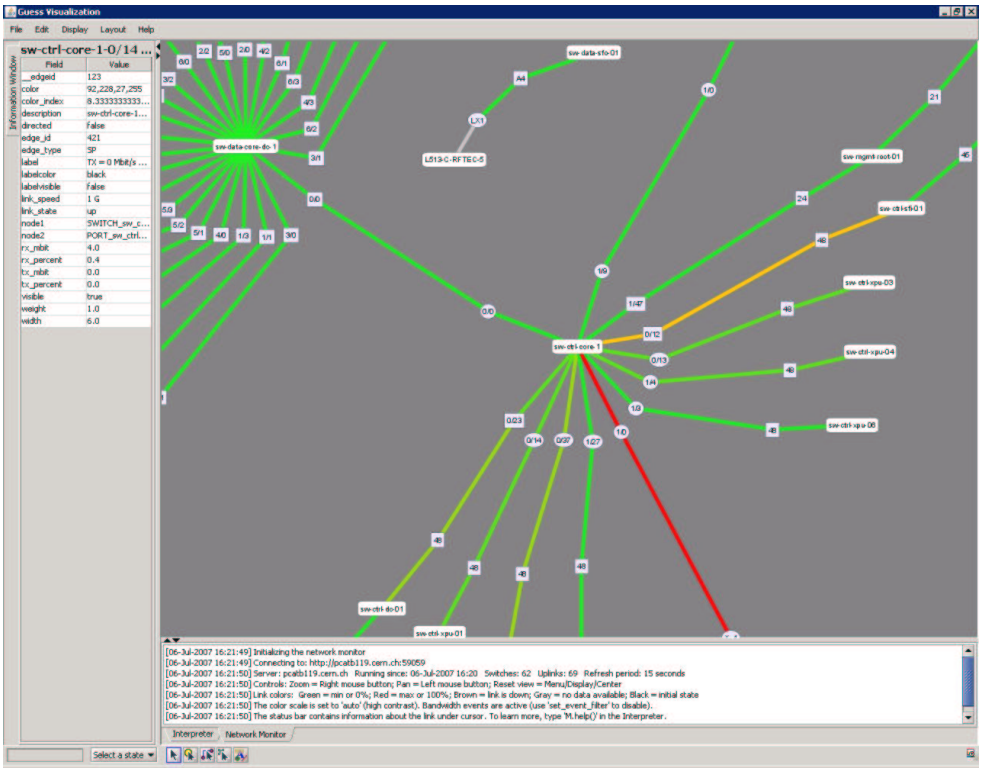


Figure 5.3: Traffic monitoring system with 2D visualization.

uses a set of hosts to inject special packets in the network. From the way these packets are seen by the other hosts, they can infer the topology. This method, like the previous one, does not require any support from the interconnect hardware. It works on enterprise switches as well as on consumer-grade devices.

One of the drawbacks of methods which are not based on MAC tables, is that they cannot provide detailed information down to the port level. That is, they cannot say to which switch port an uplink cable is plugged in, neither where exactly a PC is attached to. This makes them unsuitable for an inventory system.

5.4 Summary

In this chapter we touched two aspects of network management: device configuration and topology discovery. The tools we developed have been successfully used during the installation and commissioning phases of the TDAQ network.

We developed the *sw_script* Python module and we used it in a number of different projects for performing automatic configuration changes on devices. Then we designed and implemented an algorithm for discovering the topology of the network at Layers 2 and 3. Using the *sw_script* module and the topology discovery algorithm, we made a system that can make the inventory of all the installed equipment. By periodically running this tool, we were able to

detect cabling mistakes and track the progress of the installation (by observing the changes between two inventory reports). A traffic monitoring application has also been developed to allow the administrators to quickly see the status and the usage of the most important links in the network (the connections between the switching devices).

The next chapter covers a different aspect of network management, which is more important when the network is actually put into production. During the operation phase, changes in device configuration and in the network topology are less frequent; what matters is the performance of the whole system. We describe next a traffic monitoring technology.

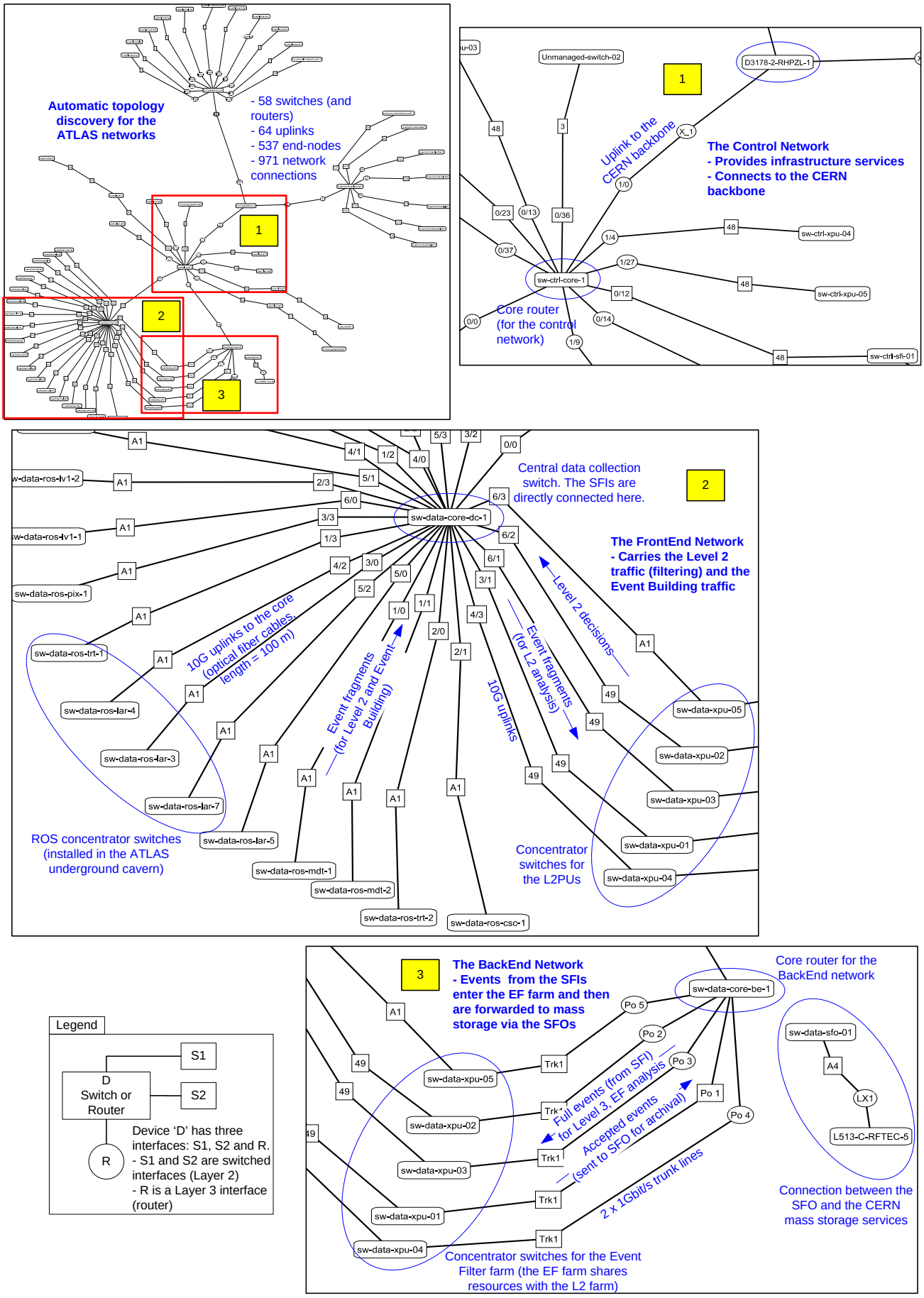


Figure 5.4: Topology discovery in the ATLAS network.

Part IV

Operation phase

Chapter 6

Traffic monitoring using statistical sampling

The performance of the ATLAS TDAQ network needs to be constantly monitored during its operation. In addition to the health monitoring (checking that all devices and all links are working), we have to watch the traffic levels in all points of the network.

The traditional technique for passive traffic measurements is based on the Simple Network Management Protocol (SNMP). SNMP allows reading the interface counters available from any switch or router. It is a simple and effective way of measuring the average occupancy of the links.

The main disadvantage of this method is that it does not offer any insight into the "contents" of the traffic. If the utilization of a link is close to 100%, there is no way to learn using SNMP what kind of traffic is taking most of the bandwidth (the "top" consumers).

One way to deal with an event like this is to effectively deviate the traffic through a *network analyzer* (sniffer) and try to understand what is causing the problem (by observing all the packets). If a hardware network analyzer is not available, another possibility is to use a feature called "port-mirroring" which is available in many modern devices. In this way the traffic can be "copied" from one switch port to another, the "mirror port" having a PC with a software *sniffer* attached [1].

The issue is that none of these methods can be deployed on a wider scale. Network analyzers are expensive and software sniffers cannot keep up with todays multi-Gigabit speeds. In addition, the amount of collected data is huge, if all packets are recorded. Other methods are needed to troubleshoot network problems in large high-speed networks.

[1] A "sniffer" is a program that can record all the packets transfered on a network interface. Examples are *tcpdump* and *Ethereal*.

6.1 Traffic sampling

In order to understand the structure of the traffic in a network and to measure traffic attributes it is necessary, in principle, to examine every packet. But as explained above, this is not always possible.

A solution is to sample the traffic stream and examine only a small subset from all the packets. By taking the appropriate packet samples, it is possible to infer the characteristics of the whole stream using statistical techniques ([56], Section 6.1.1). Even though we cannot get absolute accuracy using sampling (and then extrapolation), we shall see later that the precision of the measurement can be quantified. The samples that are taken do not need to be complete copies of the packets – for most applications it is sufficient if only the protocol headers are saved. This also simplifies the storage and transfer requirements of the samples.

As an example, we consider a line of 10 Gbit/s which transfers 1.2 Gbyte/s. If we save one packet in every 4000, we reduce the rate to 300 Kbyte/s. If we further keep only the headers (i.e. the 64 bytes from each packet), we are left with 13 Kbyte/s. This amount of data can be easily stored or further processed in real-time.

There are a number of ways that can be used to decide which packets to be considered for sampling. The easiest one is to use *deterministic sampling* and to save every N^{th} packet – unfortunately, this cannot handle periodic patterns. Another method is to use a *time-driven sampling* and select a packet when a timer expires. If the timer is initialized with a random value, this method will have good results. But the method most widely used and that we are going to discuss in greater detail is called *1-of-N random sampling*. It works by capturing *on average* one packet out of every N. The method does not use a fixed value of N, but a randomly distributed value having the mean equal to N. If N is uniformly distributed, then virtually any packet has equal chances of being selected (sampled).

The goal of sampling is to reduce the amount of data and still be able to get information about the structure of the traffic. Once the samples are taken, they have to processed in order to infer the characteristics of the original stream.

In this chapter we shall first present the theory on which sampling is based (Section 6.1.1). Then we shall describe *sFlow* which is the network industry standard for statistical sampling (Section 6.2). The second half of the chapter is dedicated to a monitoring application, based on packet sampling – implementation details and results will be presented in Section 6.3.

6.1.1 Packet sampling theory

We shall introduce now the notations and the most important results on which statistical sampling is based. An in-depth presentation of this topic can be found in Appendix C.

We assume we have a network device that during a period of time, T, transfers N packets. We are interested in finding the number N_c out of these packets which have a certain attribute C.

During time T, the network device randomly samples n packets out of the total of N. By analyzing these samples we observe c packets having attribute C. It can be shown that an estimate of N_c is:

$$\hat{N}_c = \frac{c}{n} \cdot N$$

If the number of samples (n) is large, then the relative sampling error is given by:

$$error = 196 \cdot \sqrt{\frac{1}{c}} \, [\%]$$

The estimate $\hat{N}_c$ gives the number of packets in class C. The average number of bytes consumed by this particular class is:

$$\hat{B}_c = \bar{b_c} \cdot \hat{N}_c, \text{with } \bar{b_c} = \frac{\sum_{i=1}^{c} b_{ci}}{c}$$

In the above formula, $\bar{b_c}$ is the average size of a sampled packet. The accuracy of the estimates depends mostly on the number c of "interesting" samples (the ones found in the sample set). To decrease the error, one can increase the time T or increase the sampling rate (increase n so that c is likely to increase as well).

6.2 Packet sampling with sFlow

sFlow [57] is a packet sampling technology that was designed to operate continuously on networks working at multi-gigabit speeds. It is supported by the majority of network equipment manufacturers[2] [59]. *sFlow* employs *1 of N random sampling*. It comprises two main components (Figure 6.1): one is the *Agent* and runs inside network devices (switches and routers). This component is responsible for the actual sampling of the traffic flows. The packet samples are encoded into *sFlow* datagrams that are submitted periodically to the second component, the *Collector*. This is represented by PC software which gathers the *sFlow* samples from all the agents and derives the characteristics of the traffic flows.

The *sFlow* packet samples are not complete copies of the packets. Only the headers are extracted. The packet samples are also called *flow samples* because they are used to derive information about the traffic flows present in the network (a flow can be regarded as a conversation between a pair of hosts).

[2]NetFlow [58] is another technology which is similar to *sFlow* in the sense that it collects information about the traffic flows in the network. NetFlow, however, is not based on sampling and it requires a complex implementation of the agent inside the device. Few manufacturers support NetFlow because it is a proprietary technology from Cisco.

In *sFlow*, there is minimal data processing in the Agent (hardware). The system is designed such that the analysis and data collection is done in software on the PC (where the storage space is virtually unlimited). *sFlow* uses UDP for sending the datagrams. This allows for a simpler implementation of the Agent. The system is not sensitive to losses – the effect of a few lost datagrams is a slight reduction of the sampling rate. In the following we describe the Agent and the Collector in more detail.

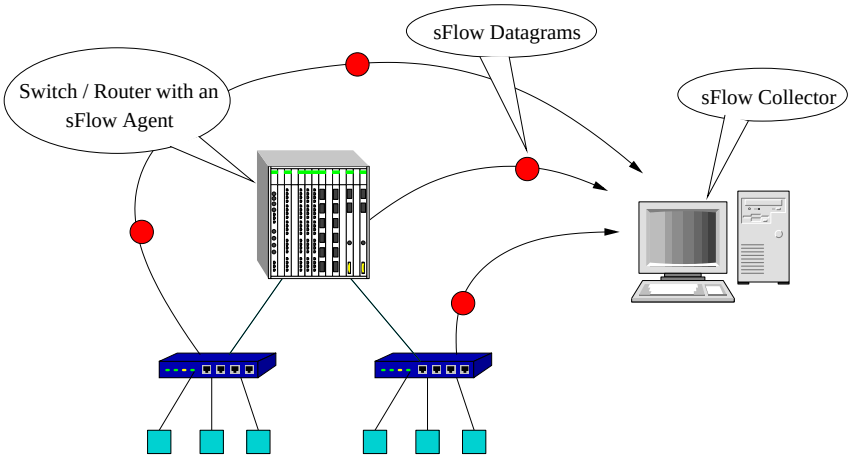


Figure 6.1: *sFlow* Agent and Collector.

6.2.1 The sFlow agent

The *sFlow* agent is implemented in hardware, usually in an ASIC[3], so that sampling does not affect the overall performance of the switching device. A packet sampler is attached to each interface of the device. Two sampling processes are running: one for statistical sampling of the packets and one for time-based sampling of the interface counters.

Statistical (packet-based) sampling of switched flows This mechanism produces *flow samples*. The process must insure that any packet involved in a switched flow[4] has an equal chance of being sampled, irrespective of the flow to which it belongs. Taking a sample involves copying the packet's header and extracting some features from it. The samples must be taken at random but *on average* there should be one sampled packet out of N (where N is a configurable parameter). The following information is saved with the sample:

- Forwarding information: source and destination physical interfaces
- Layer-2 protocol information: source and destination MAC addresses, VLAN tags
- IP headers: source and destination IP addresses, TCP / UDP port numbers, IP options
- Routing protocols information

[3]Application-Specific Integrated Circuit.

[4]A switched flow corresponds to the packets that enter on one interface of the switch/router and leave on another interface.

- A small number of bytes copied from the payload of the sampled packet (10 to 20 bytes)

Periodical polling of network interface statistics (time-based sampling) The second responsibility of the agent is to periodically send the number of packets and bytes from each data source (interface) – these are the *counter samples*. A maximum polling interval (P) is assigned to the agent, but the agent is free to schedule the polling in order maximize internal efficiency. The counter samples are needed to derive traffic estimates (to compute the total bandwidth occupancy of an interface).

The agent combines flow and counter samples into *sFlow* datagrams and sends them to the collector. Each datagram contains the identity of the agent and a timestamp. The parameters of the sampling processes can be set using SNMP: the number N which gives the sampling rate and the period P at which the counters are read. Common values are $N = 4096$ and $P = 30$ seconds. Note that N is the average number of packets that can pass between two consecutive samples. The actual value of N is a random number. In addition to N and P, the Agent has to know the address of the Collector (where to submit the samples).

6.2.2 The sFlow collector

The *sFlow* collector is the point where all the *sFlow* datagrams are received. It is responsible for the analysis of the samples in order to provide information about the original flows. Basically, it classifies and counts the packets using the flow samples and extrapolates the data using the counter samples. For example, it can filter all samples coming from a particular interface and make a histogram of IP source-destination pairs. In this way the administrator can see the how much bandwidth is allocated to each IP transaction. The collector can also be used for security applications, to detect Denial-of-Service attacks and unauthorized network usage.

6.3 An sFlow monitoring application

For a better understanding of the technology, an *sFlow* collector/analyzer has been implemented. This analyzer acts as a sink for *sFlow* datagrams and can compute various traffic statistics. In the following, we present first the general organization of a collector, we continue with a few commented results and then we discuss the accuracy of *sFlow*.

6.3.1 General architecture

We implemented the *sFlow* collector in the Python language. The use of a scripting language simplifies the development cycle and allows new features to be added easily.

The block architecture of the analyzer is illustrated in Figure 6.2. Each module is represented

by a thread in the implementation. The *sFlow* datagrams enter into the Collector module[5] which extracts the counter and flow samples and puts them into a queue. At the other end of this queue we have the Assembler block. This one accumulates samples into a local buffer until it receives a signal from the Trigger module. The signal can come either at a fixed interval of time (usually a few minutes), or after a certain number of samples has entered the Collector. After it gets the trigger, the Assembler pushes all the samples it has in its buffers to the Analyzer block (the object of this transfer is a *bunch of samples*). This is the place where all statistics are computed – the Analyzer sorts the samples by agent and then by source (physical switch port) and then identifies the flows inside each bunch. The bunches can be retrieved either one by one or can be aggregated. A bunch is associated with what has happened in the network in an interval of N minutes (where N is the time between two triggers). This data organization allows us to see the "history" and observe traffic trends.

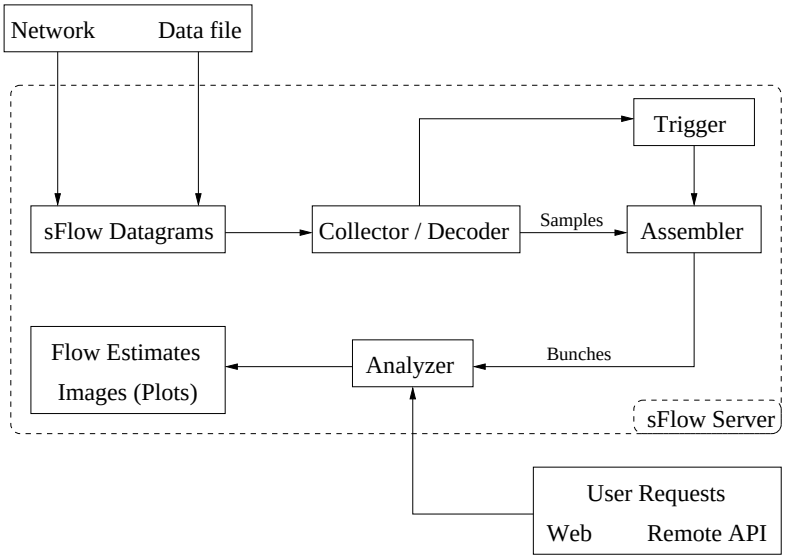


Figure 6.2: Block diagram of an *sFlow* analyzer.

The statistics and the samples can be viewed as plots[6] via a web interface or read via RPC[7] requests. The remote API allows the user to connect to the *sFlow* analyzer and use the traffic estimates in other applications.

The next sections describe in more detail the inner workings of the *sFlow* analyzer. Although we shall refer to our particular implementation, the basic principles apply to any other *sFlow* analysis package.

6.3.2 Classification of samples

sFlow-enabled devices send UDP datagrams containing samples to the *sFlow* collector. The

[5]The datagrams can be read from a file (offline) or directly from the network (online operation).
[6]Sample plots are shown in Section 6.3.4.
[7]RPC = Remote Procedure Call. Our implementation is based on XML-RPC [60].

format of these packets is defined in [57]. Listing 6.3.2 shows a decoded flow sample (left column) and a decoded counter sample (right column). We shall refer later to the fields marked with an asterisk (*).

Listing 6.3.2 – *sFlow* samples

```
--- Flow sample ---

    datagramSourceIP 10.156.0.14
    datagramSize 1224
    unixSecondsUTC 1139923089
    datagramVersion 5
    agentSubId 0
(*) agent 10.156.0.14
    packetSequenceNo 4760
    sysUpTime 1661834200
    samplesInPacket 7

    sampleType_tag 0:1
(*) sampleType FLOWSAMPLE
    sampleSequenceNo 2983
(*) sourceId 0:48
    meanSkipCount 512
    samplePool 1122083834
    dropEvents 1
(*) inputPort 12
(*) outputPort 48
    flowBlock_tag 0:1
    flowSampleType HEADER
    headerProtocol 1
    sampledPacketSize 88
    strippedBytes 4
    headerLen 84
    headerBytes 0A-00-30-0A-9C-00-00-40-9E-00-65-3F-08-00-...
    dstMAC 0a00300a9c00
    srcMAC 00409e00653f
    IPSize 70
    ip.tot_len 70
    srcIP 10.156.145.130
    dstIP 192.228.79.201
    IPProtocol 17
    IPTOS 0
    IPTTL 64
    UDPSrcPort 32770
    UDPDstPort 53
    UDPBytes 50
```

```
--- Counter sample ---

   sampleType_tag 0:2
   sampleType COUNTERSSAMPLE
   sampleSequenceNo 370
   sourceId 0:2
   counterBlock_tag 0:1
   ifIndex 2
   networkType 6
   ifSpeed 1000000000
   ifDirection 1
   ifStatus 3
   ifInOctets 658167528
   ifInUcastPkts 6012624
   ifInMulticastPkts 14
   ifInBroadcastPkts 111115
   ifInDiscards 0
   ifInErrors 0
   ifInUnknownProtos 0
   ifOutOctets 4029591497
   ifOutUcastPkts 13024900
   ifOutMulticastPkts 1705946
   ifOutBroadcastPkts 19087685
   ifOutDiscards 0
   ifOutErrors 0
   ifPromiscuousMode 0
   counterBlock_tag 0:2
   dot3StatsAlignmentErrors 0
   dot3StatsFCSErrors 0
   dot3StatsSingleCollisionFrames 0
   dot3StatsMultipleCollisionFrames 0
   dot3StatsSQETestErrors 0
   dot3StatsDeferredTransmissions 0
   dot3StatsLateCollisions 0
   dot3StatsExcessiveCollisions 0
   dot3StatsInternalMacTransmitErrors 0
   dot3StatsCarrierSenseErrors 0
   dot3StatsFrameTooLongs 0
   dot3StatsInternalMacReceiveErrors 0
   dot3StatsSymbolErrors 0
```

Once received, the samples are classified by the following criteria:

1. The *sFlow* agent which sent them.
2. The ID of the source inside the agent (which, in most cases, corresponds to a physical port interface).
3. The type of the sample (Flow sample or Counter sample).
4. The direction – if the packet was sampled while it was entering or while it was leaving the device.

The samples are stored in a data structure organized hierarchically based on the criteria above. This kind of arrangement facilitates the access to the data gathered from a particular *sFlow* source.

The "direction" is determined using the *inputPort* and *outputPort* fields. If the *inputPort* corresponds to the *sFlow* source then it means the packet was sampled while it was entering the device; otherwise the packet is classified as going out (leaving the device).

We note that the *sFlow* standard does not specify whether sampling should be done on the input or on the output path. It is up to the manufacturer to decide. The only constraint is that a packet should not be sampled twice. Some switches sample only inbound packets while others only the outbound traffic. This is why, in order to build a complete picture, one has to use the samples from *all* the interfaces.

6.3.3 Calculations

After the initial classification, the accounting process can continue with the estimation of the characteristics of the traffic. The two main goals of the analysis are:

- Estimate the link utilization, i.e. the amount of bandwidth used at each switch port.
- Find the traffic flows which are the main contributors to the bandwidth usage in the network.

6.3.3.1 Link utilization

The link utilization can be computed using the counter samples. Each sample contains the number of packets and bytes recorded by the physical interface[8] at the time the sample was taken[9]. With at least two counter samples taken by the same source at different moments, we can compute the link utilization using the formula[10]:

$$L_S[\%] = \text{Link utilization for source S} = 100 \cdot \frac{\frac{\Delta\text{Bytes}}{\Delta t} \cdot 8 + \frac{\Delta\text{Packets}}{\Delta t} \cdot 8 \cdot 20}{\text{LinkSpeed[bits]}}$$

The formula assumes that during the time Δt there were ΔPackets transferred with a total size of ΔBytes. This formula computes the number of bits used by the packets on the line. For each packet we need to add 20 bytes corresponding to the minimum interpacket gap. The utilization is expressed in percents of line speed.

[8] For both in/out directions. See Listing 6.3.2.

[9] This time is known from the *sFlow* datagram.

[10] This expression is valid only for the Ethernet networking technology. This is due to the fact that it takes into account the minimum interpacket gap (IPG) which corresponds to the time required to send 20 bytes (this gap is specific to Ethernet). The IPG is a period of silence that follows the transmission of every packet.

6.3.3.2 Flow estimates

The main purpose of statistical sampling is to find and quantify the traffic flows in the network. In general, a switch port observes and samples packets from many flows. There is no authoritative definition for a traffic flow, because the interpretation depends on the application. We shall consider a flow as a sequence of packets which are characterized by a common set of features. This set is called a *flow key*. The most common flow keys are those based on IP addresses – such a key might consist of a tuple made of: source and destination IP addresses, port numbers and protocol type (UDP, TCP, ICMP, etc.)

First, the sample set is searched for unique flow keys. Then the samples are classified by their keys. At the end, the number of packets and bytes is estimated for each class. We list below how we've chosen to categorize the packets in our implementation:

IP packets For such packets we consider that their source and destination IP address represents the flow key. An extension would be to use also the IP protocol and the UDP/TCP port numbers as part of the flow key, in order to differentiate among the flows belonging to different applications.

Broadcast and Multicast packets These are packets having a broadcast or a multicast address as their destination. Their flow key contains the MAC address of the host which sent the packet.

Flooding As we have seen before, any flow sample contains a field called *outputPort*. Normally, for unicast messages, this field contains the id of another port on the device. If a packet has to be forwarded to more than one interface (multiple destinations), the *outputPort* will have a special format which encodes the number of ports that get a copy of the packet. For broadcast and multicast packets this is the expected behavior. However, if we find a flow sample containing a unicast destination MAC address and which is being copied to all ports, this means that the packet is *flooded*. Any switch will proceed to flooding when it doesn't know how to forward the packet – after one such packet, the correct recipient should reply, the switch learns its address and no longer does flooding. If there are a lot of flooding packets detected by the *sFlow* analyzer, then this is a sign of a serious malfunction in the network. Packets belonging to a flooded flow are recognized and categorized separately; their flow key contains the source that generates the flooding.

The following formulas are used to compute the bandwidth used by each flow.

Number of packets	$N_c = \frac{c}{n} \cdot N$
Number of bytes	$B_c = \overline{b} \cdot N_c$
Link utilization	$U_c = B_c \cdot 8 + N_c \cdot 20 \cdot 8$

c **- The number of samples in a class** This is simply the number of flow samples that have the same flow key. For example all packets from host A to host B might be considering as part of the same class.

n - **The total number of samples** This is again easily determined by counting all samples from the same *sFlow* source (having the same *sourceId*).

N - **The number of packets that could have been sampled** Each flow sample contains a field called *samplePool*. This number counts how many packets passed through a source/port and had the opportunity of being sampled. If we have a sequence of flow samples from the same source, we can simply make the difference between the first and last values of the *samplePool* and determine the value of N.

b - **The average size for the packets in class** c As the name suggests, the value of b is computed by averaging the value of the field *sampledPacketSize* from each flow sample.

At this point we have the utilization U_c for each flow identified from a switch port S. The total link occupancy, L_S, can be used to express U_c as a percent of the line speed.

6.3.4 Examples

The next paragraphs show what kind of output can be obtained from an *sFlow* analyzer. We shall use the network presented in Figure 6.3 as a basis for our discussion. This network consists of three switches which are interconnected by two uplinks: between S1 and S2 there is a 1G connection, while S2 and S3 are connected by a 10G line. Each switch has a few end-nodes connected to it – these nodes can be configured to send IP traffic at a given rate to an arbitrary destination. The nodes are in fact GETB tester ports (Chapter 3).

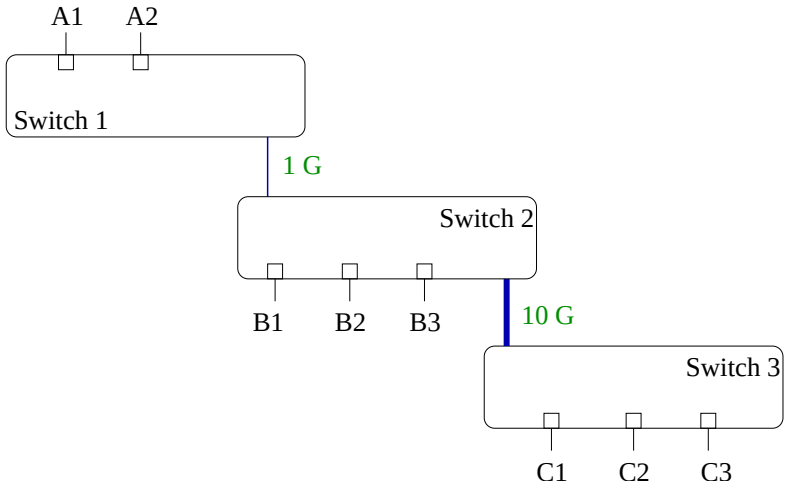
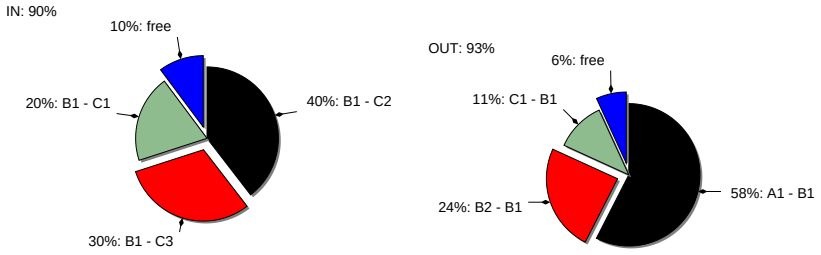


Figure 6.3: Network used to test *sFlow*.

6.3.4.1 Pie charts

The most basic functionality of an *sFlow* analyzer is the ability to reconstruct the traffic profile for a source. Not only we can see how much traffic a host is sending and receiving, but we can also

see where the packets are going and where are they coming from. Figure 6.4 shows a graphical representation for the traffic passing through an *sFlow* source. The two pie charts are associated to one port of a switch (in this case it is switch S2 and port B1). The traffic is inspected for each direction separately. Note that what is seen as *input* by the switch is being *sent* by the host (B1 in this case). The "out" from the switch refers to what is *received* by the computer.



(a) Input to the switch port B1

(b) Output from the switch port B1

Figure 6.4: Sample traffic profile – Pie chart for port B1 on switch S2.

By looking at the plot we see that node B1 uses 90% of the line capacity to send data to the remote nodes C1, C2 and C3. We also see that B1 receives data from A1, B2 and C1, most of the input bandwidth being allocated to the traffic from A1 ($\approx 60\%$).

6.3.4.2 Traffic matrices

Pie charts like in Figure 6.4 can be obtained for every switch port. In order to get a global picture of what is happening in the network, a different representation is preferable. A type of graphic that can convey a lot of information is the *traffic matrix*. This describes in a compact form the traffic between a set of hosts. In this representation we associate a matrix row for each possible transmitter (sender) and one column for each possible receiver. A cell $A(i, j)$ on row i and column j, contains the amount of data that is *sent* from node i to node j. We express this quantity in percents of line speed[11].

The above description refers to a TX traffic matrix – which tells us how much data the end-nodes transmit into the network. Similarly, we can define the RX traffic matrix. When there are no losses inside the network, the RX matrix is simply the transposed of the TX matrix.

Using *sFlow* we are able to measure both types of matrices, the TX and the RX. We've found, however, that the estimation for the TX matrix is more accurate than the RX. This happens for the RX matrix even in the absence of packet loss.

As an example, we configured the nodes from Figure 6.3 to send traffic between them and we used *sFlow* to reconstruct the traffic matrices.

[11] We assume that the link speed is known for all nodes connected to the network.

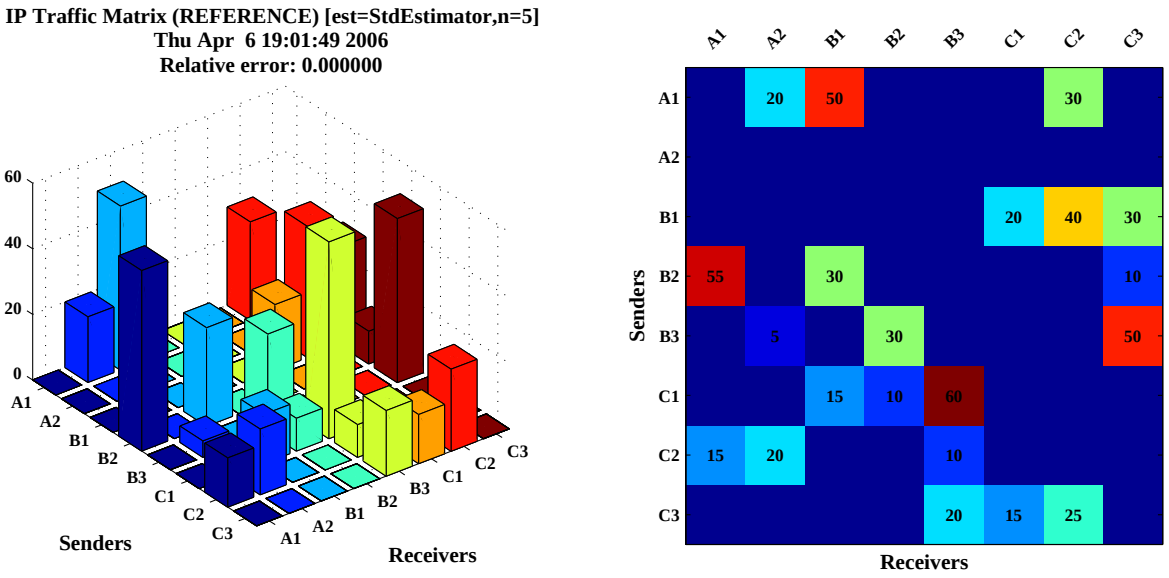


Figure 6.5: Reference traffic matrix (input to the GETB tester).

In Figure 6.5 we show the "reference" TX traffic matrix, which is in fact the traffic description that is the input of the GETB system. The TX matrix has been chosen such that there should be no packet loss. The matrix is shown as a 3D bar chart and as a 2D image map. The values in both representations are expressed in percents of line speed. The row corresponding to B1 contains the same information as Figure 6.4(a).

Figures 6.6 and 6.7 reveal the TX and the RX traffic matrices as they were reconstructed using *sFlow*. Each figure contains also the relative error computed with respect to the reference matrix. The estimate for the TX matrix is much more accurate than the one for RX. A possible explanation is that the switches we used for our experiments tend to sample the packets as they enter the device[12]; if this happens, then the estimation for the packets going out (what is received by the end-nodes, the RX matrix) will be less accurate (less samples recorded).

This example used artificial traffic composed of packets having the *same size*. In the next section we study how the estimation is affected by the presence of a mixture, containing packets of different sizes.

6.3.5 Accuracy

The previous example used a complex network to show what kind of results can be expected using statistical sampling. A simpler example is shown in Figure 6.8 in which we have a source A sending to two destinations B and C. The amount sent to B increases from 10% to 80% (*T B(rx)* in the legend) and the amount sent to C decreases from 80% to 10% of line speed (*T C(rx)*). At any moment their sum is 90%. During the test we modify the loads by 10% at each iteration

[12]What enters the switch is transmitted by the end-node. The TX matrix refers in fact to what is *received* by the switch from the outside.

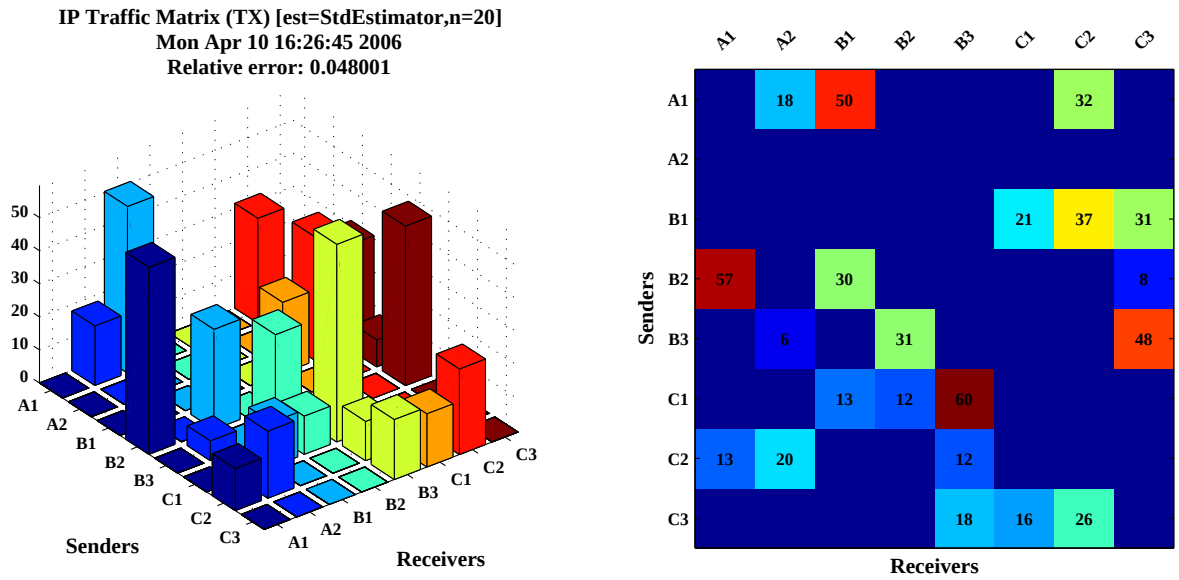


Figure 6.6: *sFlow*– TX traffic matrix.

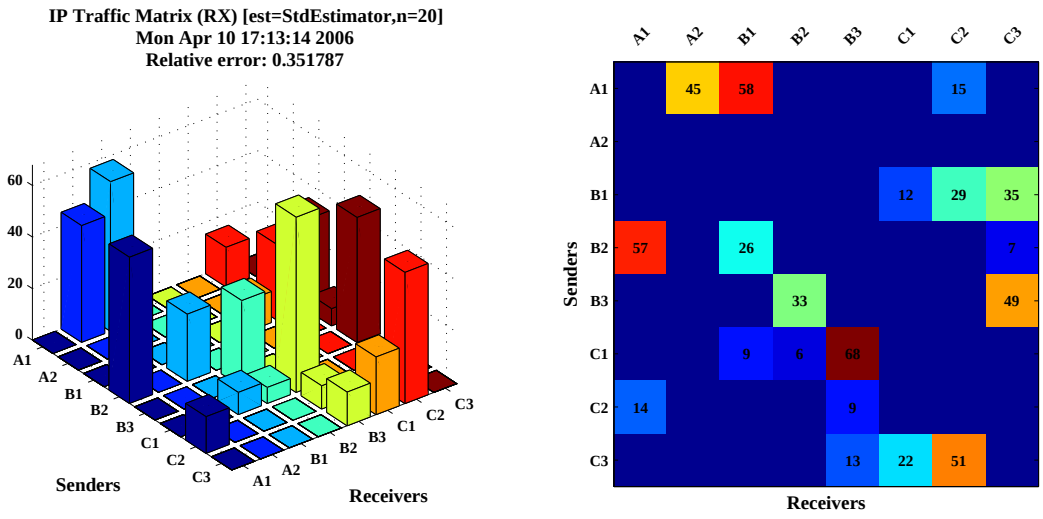


Figure 6.7: *sFlow*– RX traffic matrix.

and measure using *sFlow* the loads at the transmitter (*S AB(tx), S AC(tx)*) and at the receiver (*S B(rx), S C(rx)*). It can be seen that the estimates are quite accurate in this case. Note that all packets have (again) the same size.

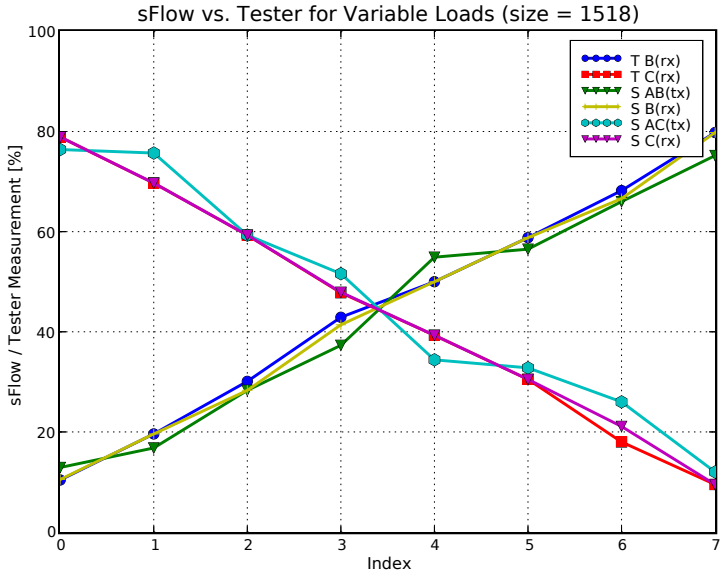


Figure 6.8: Estimation of two flows sharing a link (constant sum, variable ratio).

In Section 6.1.1 we saw that the accuracy of the *sFlow* estimation depends on the number of samples. As the decision whether to sample a packet does not take into account its size, it is likely that a flow that will consist of many small packets will be sampled more often than one composed of large packets.

The calculation of the effective line utilization for a flow takes into account the average packet size from the sequence of packet samples. From the formula for the utilization (U_c, page 149) we see that U_c is proportional to the product between the average packet size and the number of samples observed from that flow ($U_c \propto b \cdot N_c \propto b \cdot c$). For two flows that we know "a priori" that use the same bandwidth we should have $b_1 \cdot c_1 \approx b_2 \cdot c_2$ or $\frac{c_1}{c_2} \approx \frac{b_2}{b_1}$.

6.3.5.1 Under-sampling

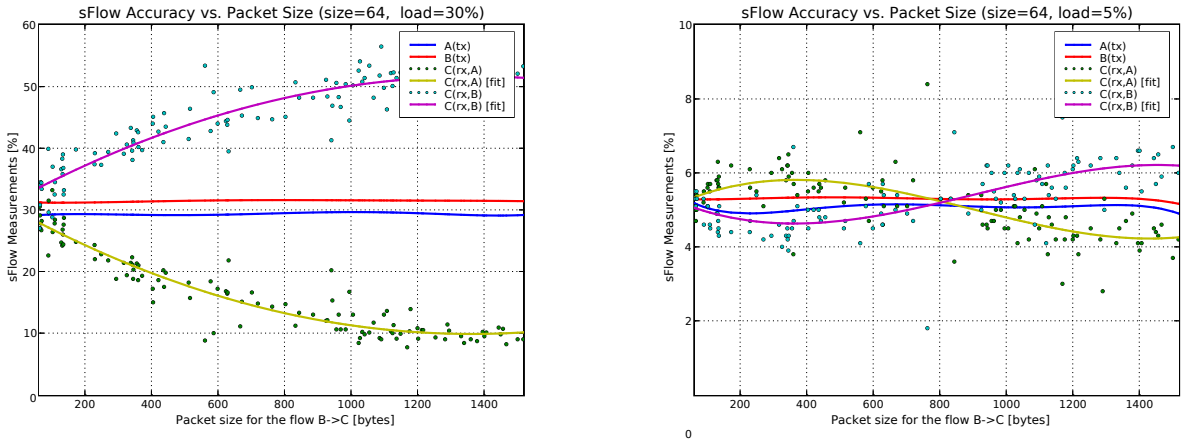
We present now an example when the *sFlow* agent cannot sample packets fast enough. Because of this, some flows will be under-sampled and, consequently, their bandwidth utilization will be underestimated.

Suppose we have two flows that traverse an *sFlow* sampler. Flow 1 contains only small packets (64b), while flow 2 only large packets (1518b). We know that both these flows are using the same amount of bandwidth. Then, in order to get a good estimation, the *sFlow* sampler must take enough samples from flow 1, c_1, such that the ratio $\frac{c_1}{c_2}$ equals $\frac{b_2}{b_1} = \frac{1518}{64} = 23$. If the *sFlow* engine is not able to keep up with the rate and will drop samples from flow 1, then we shall observe

an overestimation of flow 2 and an underestimation of flow 1 (because we'll have $\frac{c_1}{c_2} < \frac{b_2}{b_1}$ and so $U_{c1} < U_{c2}$).

In order to show that this is really the case, we made the following test. We configured two nodes A and B to send to the same destination C, each one sending at the same load $L = 30\%$. Node A sends packets of a constant size $b_1 = 64b$, while node B sends them with size $b_2 = x$ (but at the same load L). We vary the size x and at each iteration we measure the estimate produced by *sFlow* at node C. We shall measure the flows $A \rightarrow C$ and $B \rightarrow C$ as they are received at C (so we use only the sampler embedded in switch port C).

In Figure 6.9(a) we show how this under-sampling effect affects the accuracy of the measurement. It can be observed that the error increases with the difference between the two sizes (from sources A and B). The flow $A \rightarrow C$ is largely underestimated at the far end of the plot, when the flow $B \rightarrow C$ uses a packet size of 1518 bytes. In Figure 6.9(b) we performed the same test, but now using a different load L = 5%. The errors are clearly smaller in this case, as the *sFlow* sampler can keep up with the lower arrival frequency of packets.



(a) Load L = 30%

(b) Load L = 5%

Figure 6.9: Estimation of two flows: $A \rightarrow C$: 64b @ load L, $B \rightarrow C$: x bytes @ load L.

The effect we observed seems to be due to the limited processing capacity of the *sFlow* sampling engine inside the device we used for testing. Increasing the sampling rate will not help in such a situation as the limitation is in the hardware itself. In fact one possibility to reduce the errors would be to decrease the sampling rate so that the device has more time at its disposal.

Fortunately, in real networks the traffic is composed of packets of many different sizes so it is unlikely to observe the issues we've just described, related to the accuracy. And with *sFlow*, the precision gets better as the sample collection period increases.

6.4 Future applications

Packet sampling is likely to become the most popular method for traffic engineering in the future high speed networks. Some areas which may benefit from the use of a sampling technology like *sFlow* are the following:

- *Congestion management.* By monitoring traffic flows on all ports continuously, *sFlow* can be used to instantly highlight congested links, identify the sources of the traffic, and the associated application-level conversations. Then it can trigger actions to alleviate the congestion: rate-limiting some traffic flows, prioritizing others or simply informing the network administrators.
- *Trending and capacity planning.* The analysis of the detailed traffic reports obtained using packet sampling, allows the determination of the most demanding flows and consequently can help adapt the network topology or routing to better suit these demands.
- *Security applications.* Packet samples containing complete headers (layers 2 to 7) can be used to identify security threats. Intrusion detection can be easily implemented and patterns of Denial of Service attacks can also be discovered.

sFlow agents are now embedded in the products of most network equipment manufacturers. *sFlow* collectors are available both as commercial products and open-source programs [61].

While the basic function of *sFlow* is to obtain per-flow statistics, we think the technology could be adapted for other applications as well:

Finding the topology using *sFlow* We've seen in Section 5.3 an example of an algorithm that can find the Layer-2/3 topology of a network, if the MAC address tables are known. There are situations when these tables are not accessible. Then we can use *sFlow* samples to build at the collector the Layer-2 or Layer-3 address mapping tables. The algorithm will need more time to acquire the data, but it would arrive to the same solution (network map).

Predicting packet loss – Finding the hot spots A potential application of *sFlow* is the prediction of the hot spots in the network[13]. Knowledge of the network topology allows the prediction of the intended load in any point inside the network, once the *global* TX traffic matrix is available. This requires *sFlow* to be enabled only at the edges of the network (near the end-nodes[14]). The TX matrix gives the information about the network inputs, while the network map tells us how the data travels. Using these two, the load exerted in any point (uplink or output port) can be calculated. The cumulative load on any uplink can be predicted. The main idea is that the utilization of the uplinks and of the output ports can be estimated and any overflow can be promptly reported.

[13]Hot spot = a switch port where congestion has been created and packet loss is likely to occur.

[14]It is possible to collect *sFlow* samples right from the network card of the PC. This affects the performance, but can be used in networks containing devices without support for *sFlow*.

Checking the efficiency of the ATLAS TDAQ system In Chapter 2 we've explained that the events captured by ATLAS are assembled from the fragments recorded by the sub-detectors. The fragments of the events that pass the Level 1 tests, are stored in the Read Out Buffers. The data from the ROBs is forwarded only on request from the Level 2 or Level 3. The traffic distribution across the ROBs (ROSs) depends on the Regions of Interest determined by Level 1. It is expected that some ROBs will be asked for data more often than others, simply because they are connected to sub-detectors that are more useful for analysis. Computer models have been created for ATLAS and physicists know what kind of distributions to expect (i.e. the utilization at the level of individual ROSs).

A useful application of *sFlow* would be to determine the traffic profiles at the output from the ROSs to the Level 2 and Level 3 subsystems. These measurements, which can be done in real-time (while the experiment is running), would provide information about the efficiency of the different sub-detectors and would tell the physicists if the system behaves according to their predictions.

6.5 Summary

In this chapter we introduced the concept of statistical sampling for monitoring the traffic in a network. The *sFlow* standard has been described and then the monitoring application we implemented has been presented. Some of the caveats that may arise when dealing with statistical sampling have been discussed, based on results obtained using the custom system we've developed. Because of its ability to monitor traffic at multi-Gigabit speeds and because of its simple hardware implementation, we expect *sFlow* to become increasingly popular in the networking industry.

All the devices we have acquired for the final TDAQ network support *sFlow*. Preliminary tests have shown that their implementation of the standard does not exhibit the under-sampling effects we've just described. As future plans, we intend to fully integrate *sFlow* tools in our network management infrastructure.

Part V

Epilogue

Chapter 7

Conclusions and future work

In this chapter we summarize the work that has been done and we highlight the contributions of the author to the development of the ATLAS TDAQ network. We finish with a few ideas of future research in the field of network management.

7.1 Summary of contributions

The TDAQ network is a critical component of the ATLAS data acquisition system. Many years of research have been allocated in order to provide the experiment with a complete and reliable solution. We outline below our contributions to this research.

7.1.1 Design of the TDAQ network

The design phase of the TDAQ network (presented in Chapter 2) began with a survey of the available technologies. Soon it became clear that Ethernet is the best solution, not only from the technical point of view (performance), but also because of its cost effectiveness and long term support. The next step was to define an architecture of the future network – although the author has contributed to this part ([16], [18], [19]), it was not his main activity. The particular nature of the TDAQ leads to a specific traffic pattern, which in turn determines particular requirements for the network equipment. After identifying a meaningful set of features for the TDAQ network devices [4], we entered a phase during which we had to evaluate and finally choose the devices that would best suit our needs. This was the area to which the author brought a major contribution, as described next:

Creation of a platform for testing network devices

We developed a system for network testing. The GETB FPGA-based platform (Section 3.2) and its main application, the network tester (Section 3.3), have been successfully used to assess the products from various vendors – both in terms of performance and functionality. In the end, the

best suited devices have been chosen, based on our recommendations. The author designed and implemented the core FPGA firmware and the control software of the GETB system. Based on his work, two other projects have been completed (see Sections 3.5.1 and 3.5.2).

The author was the main designer and developer of the GETB tester. The tester makes use of the inherent parallelism of an FPGA implementation in order to deliver full Gigabit line-speed traffic generation. It has a flexible way of defining the traffic patterns, including the ATLAS-specific request-response pattern. The tester can measure throughput, loss and one-way delay in real-time and it can also capture packet traces for offline analysis (very useful for understanding traffic patterns). All the features of the GETB tester can be programmed using the Python scripting language. An interactive command-line environment is supported together with a graphical user interface for displaying statistics.

The author implemented also all the procedures described in the testing methodology document [4]. The entire test suite can be executed with minimal user intervention, thanks also to the *sw_script* module (see below) which allows us to configure and control the device under test while a test is running. The GETB system facilitates the creation of reports with its automatic plot generation features. Using the GETB testing system we checked the performance and the functionality of more than 18 devices produced by 7 different manufacturers.

Study of the ATLAS traffic pattern

The GETB allowed us not only to check the capabilities of the devices, but also to study in detail the traffic pattern induced by the ATLAS request-response protocol. In Chapter 4 we derived analytical formulas for the rates and queues occupancies that develop under ATLAS conditions. The expressions reveal the dependency between the network conditions (buffering capacity and delay) and the parameters that characterize the ATLAS traffic (the watermarks for tokens). Our model has been validated using experimental measurements.

For the main part of this study we used the GETB tester (a hardware-based traffic generator). For a pure software perspective, we developed a set of applications based on the ATLAS communications libraries (Message Passing libraries). The *mpNetPerf* programs can be used to provide an upper bound for the level of performance attainable in the entire TDAQ system. They can be easily deployed in the entire TDAQ network and then they can be used to check if the network performance meets the baseline requirements.

7.1.2 Installation and commissioning

As the design phase was coming to an end, the acquisition of the equipment commenced. Soon after that, we initiated the actual installation of the network. During this phase, we developed tools to ease the management of the devices and to automatically check the network connections.

Automatic configuration of network devices

The hundreds of devices that will be installed in the TDAQ network will require initialization and then configuration updates. The author developed tools to ease the job of the network administrators by allowing them to write programs that can "intelligently" configure the devices (Section 5.2).

We developed *sw_script*, a Python module which provides an abstract, object-oriented, way of communication to network devices. It allows the user (network administrator) to write programs that perform changes or simply monitor a set of heterogeneous devices. *Sw_script* has been used for running the GETB tests and for configuring switches in the TDAQ network.

Tools for discovering the network topology

In order to validate the installation and quantify its progress, we developed a system that discovers the physical topology of the network, including the connectivity of the end-nodes. The system has been successfully used to document the installed network and to confirm its proper layout (Section 5.3).

The author designed the topology discovery algorithm and then implemented it using the *sw_script* module as a low-level communication interface. Our algorithm discovers the network topology up to layers 2 and 3 of the OSI model. A complete network inventory system has been developed – this tool scans the network and generates a report with all the devices and their inter-connections. By comparing two reports we can track the eventual changes in the network. The network map generated by the discovery has also been used to implement a real-time traffic monitoring application. This allows the administrators to quickly and intuitively see the status and the usage of the most important links in the network (the connections between the switching devices).

7.1.3 Operation phase

At the time of this writing, the commissioning of the ATLAS network is close to completion, and preparations are on their way for the final operation phase. Monitoring network performance will become a major part of our activity. The author carried out a study on statistical sampling, a technology used for detailed traffic analysis.

Development of an *sFlow* analysis package

In Section 6.2 we described the *sFlow* standard which is based on statistical sampling. The author developed a prototype application for the analysis of *sFlow* data (Section 6.3). We've presented the type of results that one can obtain using statistical sampling. Using our *sFlow* analyzer and the GETB tester, we've highlighted the limitations that some devices have in their *sFlow* implementations. Our study has shown that the results obtained using *sFlow* can be very useful for understanding network usage and for troubleshooting traffic-related issues (congestion).

7.2 Future work

Due to the significant complexity of ATLAS and its DAQ system we expect that, at least in the first few months, there will be "issues" that might interfere with the normal data taking. These can be caused by anything ranging from human errors up to power outages. Sometimes, diagnosing such problems can be a real challenge. With all the redundancy that was built-in by design, parts of the TDAQ network may fail or may not operate properly. In order to help the administrators troubleshoot network-related problems and distinguish them from other system-wide issues, we plan to create an "expert-system". This system could use the knowledge acquired from multiple sources in order to provide hints about the nature of an error. It would have to be tightly coupled with the network management and monitoring system. We list below a few sources of data that could be used:

Network topology This is important for root cause analysis (which solves connection related problems). The topology allows the expert system to know exactly the path of the packets through the network.

SNMP data A lot of information can be obtained by reading the SNMP databases. The state of the links, their utilization, unexpected events in switches – all these can be retrieved using SNMP.

Drop counters inside switches When we suspect that data is lost in the network, the first place to look at is at the level of switches and routers. Modern devices keep track of the packets they discard.

***sFlow* traffic profiles** When trying to understand abnormal traffic patterns, *sFlow* becomes invaluable. It is easy to find the applications contributing to a hot spot using the samples provided by *sFlow*. This would be the first action to take when packet loss is detected – finding the most demanding traffic flow, the one that contributes the most to the congestion.

In addition to these, the expert-system could access the configuration of the TDAQ and try to understand what applications are using the network and how. Information gathered from the end-nodes could also be useful: CPU utilization, memory consumption, list of processes, etc. The expert-system should be able to correlate all these data sources. If needed, it could also run special diagnostic programs to acquire more information.

Another direction of future work is in the area of network visualization. Displaying information about the state of a large network like the TDAQ is a subject of ongoing research. Two approaches are currently being investigated. One is based on the idea of representing the network as a 2D graph. The second approach aims in building a 3D interactive environment for the entire TDAQ system. The 2D graph-based view emphasizes the relationships between the network elements. On a 2D map one can see quite easily the network path used by a certain traffic flow – this makes it more suited to network administrators. The 3D representation will highlight the relations between the components of the TDAQ system, making it more comprehensible for the ATLAS operators.

Part VI

Appendices

Appendix A

Basics of Ethernet and TCP/IP

This appendix contains a brief introduction to computer networks – for an in-depth presentation we recommend Tanenbaum's book on this subject [62].

A network consists of a set of computers which are interconnected using a telecommunication system for the purpose of sharing resources and exchanging messages. The transmission medium can be shared, as in the case of wireless networks or can use point-to-point links, as in the case of Ethernet.

Generally, data is transmitted between hosts as a collection of bytes called a *packet*. Networks which transmit data in this way are called *packet-switched* networks.

Depending on their size, networks can be broadly classified into *Local* and *Wide* Area Networks (LANs, WANs). Local networks are installed in buildings or campuses. They cover an area of hundreds of square meters. Wide area networks span over the entire globe.

The speed or bandwidth of a network segment is measured in bits per second. Common speeds are 1 Gbit/s or 10 Gbit/s for Ethernet. Wireless networks work at lower speeds: 11 or 54 Mbit/s. A network may have segments running at different speeds.

A.1 Layers

In order to reduce the design complexity, network functions are broken into layers. Standards exists to define the *interfaces* between the layers, but not their implementation. The OSI[1] model is the most commonly used – it defines 7 layers: Physical, Data link, Network, Transport, Session, Presentation and Application. The upper layers use the services of the layers below them.

The first layer is concerned with the electronics and the signaling over the transmission medium. The "Data link" layer packs the bits into packets and performs the first level of integrity checks (for errors). It takes care of the data transmission between adjacent nodes in a local network.

[1] Open Systems Interconnect.

The "Network" layer handles the routing of the packets – it does its best to transfer packets from their source to their final destination, no matter how many intermediate nodes are in-between.

The "Transport" layer provides the upper layers with an error-free point-to-point communication channel (logical circuit). It handles retransmission of packets if necessary.

The demarcation between the last 3 layers is not so strict – generally they are implemented in the operating system (OS) and in end-user applications running on the computer[2]. The user application calls the services of the OS to open "communication sockets" and send "messages". The messages are divided and encapsulated into packets and then delivered to the network.

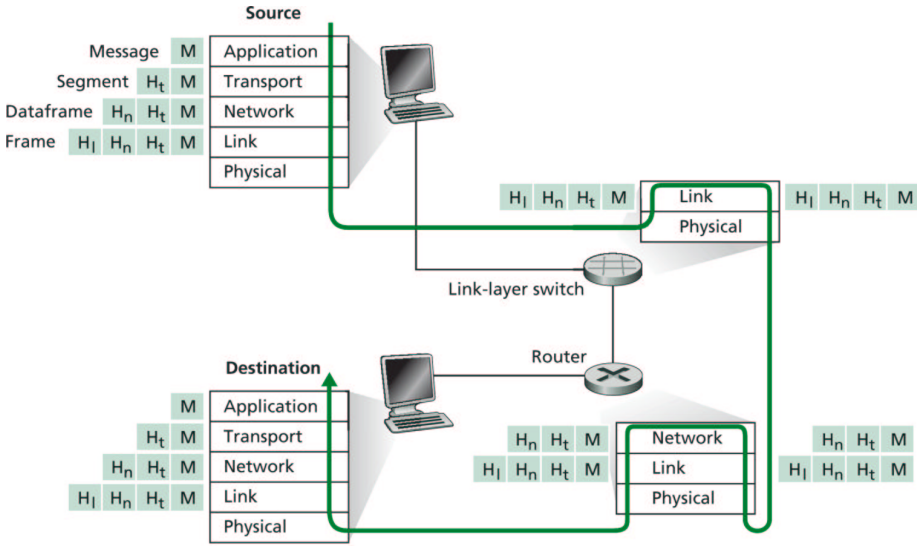


Figure A.1: Data encapsulation.

Each layer has its own way of representing the data. When a message is transmitted, each layer packs the data received from the upper layer into a "digital envelope". The envelope consists of a header and possibly a trailer. The header has the addressing information, while the trailer contains data for error recovery/detection. Normally the software/hardware operating at layer N does not modify or use in any way the data from the layers above it, $N+1$. When a message is received, it has to traverse all layers from bottom to top – this time each layer will strip the data added by the peer layer at the other end.

Network devices do not need to implement all the seven OSI layers. Only the end-nodes do (the computers). The devices that facilitate the communication over the network work only at the level of the first two or three layers. Layer-2 devices are called switches. Routers are more advanced and operate at Layer-3.

Figure A.1 shows how a message M travels from a source to a destination, passing on its way through a switch and a router (image source: [63]).

[2]The first two layers (Physical, Data-link) are implemented in hardware on the Network Interface Card (NIC). The Network and Transport layers are generally implemented in software in the OS, but they can make use of hardware accelerated functions.

A.2 Ethernet

Ethernet is the most common Data Link (DL) technology. It is used in LANs. The Ethernet packets are forwarded between hosts by *switches*. These are devices that use the addresses[3] from the Ethernet header and transfer the packets to the correct destination. An Ethernet LAN can be partitioned into *virtual* networks called Virtual LANs (VLANs). Each VLAN has an associated numeric tag. The packets inside the same VLAN must carry the same tag. The partitioning is enforced at the level of the switches. By default, they will not allow packets from one VLAN to be forwarded to another VLAN.

An Ethernet network is built like a tree. The leaves of the tree are the computers and the inner nodes are the switches. Ethernet does not allow loops inside the network graph. This reduces the complexity of the switches. They work based on the assumption that there is a single path to a certain destination. They learn these paths gradually, from the traffic they receive and have to forward – Appendix A.6 has more information.

A.3 TCP/IP

IP (Internet Protocol) is the de-facto standard for the Network layer. At the level of a LAN, the IP packets are encapsulated into Ethernet frames. At the boundary of the LAN, the Ethernet headers are removed. Packets in IP networks are forwarded by *routers*. Like the switches, they are devices with multiple network interfaces and, depending on the IP addresses[4], they know how to forward the packets to the destination.

Routers keep the forwarding information in *routing tables*. Because a router is usually located at the crossing point between a few LANs, it is not efficient to store in the routing tables the addresses of each end-node (there can be many thousands). Instead, routers learn how to forward packets to entire *ranges of addresses* – these are called *sub networks* or *subnets*. This makes the routing more efficient – a subnet is defined by the start address and the length (number of consecutive addresses).

An IP network can contain loops. Routers are aware of the possible paths to a destination network (subnet) and they always forward the packets on the optimal route[5]. Routing protocols are employed by the routers to send notifications about the availability of IP routes.

IP provides two transport protocols: the User Datagram Protocol (UDP) and the Transmission Control Protocol (TCP). UDP is a connection-less protocol. This means that UDP messages are completely independent from the network point of view (even if they belong to the same logical conversation). UDP datagrams may be lost or re-ordered by the network – the end-user application is responsible for handling such anomalies.

TCP provides an end-to-end virtual circuit (connection-oriented protocol). The TCP layer

[3]The MAC (Media Access Control) addresses. These are unique 48 bit identifiers.

[4]An IP address is represented by a 32 bit number. An extension of the IP protocol, IPv6, has 128 bit addresses.

[5]The optimal route can be chosen based on a number of factors: level of utilization, cost of the service, reliability, physical distance, etc.

makes sure that messages are delivered safely and in the order they were sent. It uses retransmission when packets are lost.

A.4 Congestion handling

Network congestion appears when a node cannot absorb all the traffic that is heading to it. The "node" can be a computer or a port on a router or a switch. In the case of network devices, congestion appears when multiple streams target the same destination. Congestion can be resolved in two ways: by losing the excess traffic or by buffering the data that cannot be handled as fast as it arrives. Congestion can also be prevented. Ethernet has a mechanism called "Flow-control". When a node cannot handle all the incoming traffic, it issues special messages ("pause" frames) to the transmitter, informing it that has to slow down the rate. This mechanism works only on point-to-point links and in some cases it lowers the rate when it is not necessary. If a traffic flow has to traverse a few switches, then Flow-control will not prevent congestion (because the "pause" messages are not forwarded by switches).

In an IP network it is the TCP protocol that handles end-to-end congestion. TCP is designed to avoid the congestion – it uses an adaptive mechanism that measures the available bandwidth and sends only at that rate. When congestion appears, TCP lowers automatically the transmission speed down to a safe value.

A.5 Monitoring and diagnostics

In its simplest form, network monitoring deals with the health status of network devices and the bandwidth usage between various network elements. The Simple Network Management Protocol (SNMP) is the standard that defines how network devices are monitored and what metrics the devices need to provide on request. Using SNMP, it very easy to find out which devices are working and which are not, the active network connections and the bandwidth occupancy on any link. SNMP is very effective in spotting problems. However, it is not as useful in diagnosing network problems, especially those related to the actual traffic pattern (see Chapter 6 for details).

A.6 Ethernet switching

Ethernet networks employ tree-like topologies built using interconnect elements called *switches*. Applications running on the end-nodes use protocols like TCP/IP for reliable communication. These high-level protocols pass the data over to the Ethernet layer which packs it into small chunks called *frames*. Any node that is connected to Ethernet and that has the capability to transmit and receive frames must a have a unique identity. For Ethernet this is represented by a 48-bit number called the *MAC address*. A frame sent to the network contains complete addressing information – namely the MAC address of the originating source and the one of the final destination.

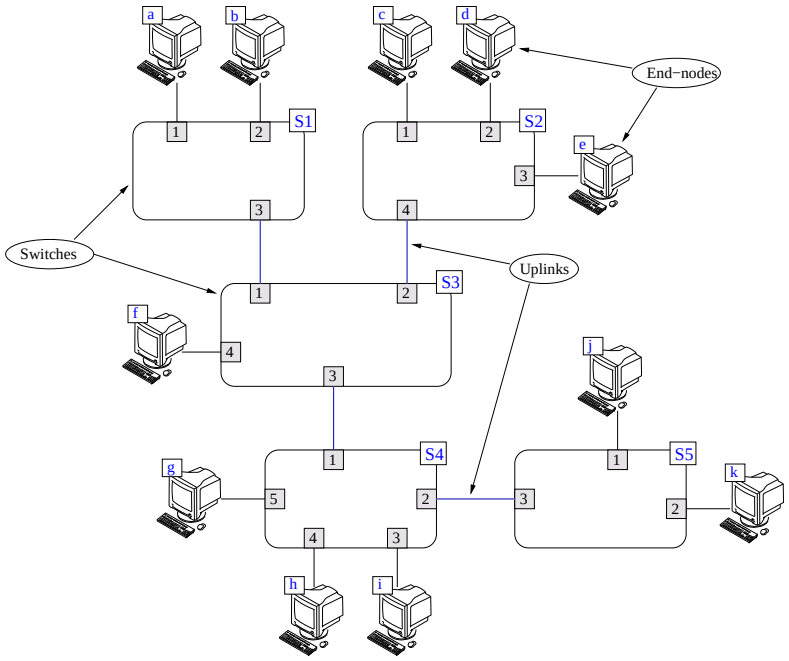


Figure A.2: An Ethernet network with end-nodes and switches.

A.6.1 Forwarding frames

Figure A.2 shows an example of an Ethernet network. End-nodes are directly connected to switches, while switches can be inter-connected between them[6] to form a tree. Switches are devices with multiple ports; their basic function is to forward frames from one port to another. Upon reception of a frame, the switch checks the destination address. For a "local" destination (connected to the same switch), the frame will be forwarded directly. If the destination is not local, then the switch will send the frame over an uplink to the nearest (neighboring) switch that *knows* how to take the frame to the final destination.

A.6.2 MAC address tables

The switches know where to send the frames by using the knowledge acquired in their *MAC Address Tables*[7]. The MAC table contains records for the mapping between node identities and the ports by which these nodes can be reached. When the switch gets a frame, it looks at the *destination* MAC address (we shall call it X) and performs a look-up in the MAC table to see which port knows about destination X. It may find in the table and entry saying "all frames to destination X should be sent via port A of the switch". If such an entry is found, then the frame will automatically go to port A[8].

[6] In the following we shall refer to the connection between two switches as an *uplink.*

[7] MAC tables are sometimes called *Forwarding Databases* (FDB) or *Address Forwarding Tables* (AFT).

[8] If the switch does not know where to send a frame, it will copy it to *all* of its ports – this behavior is called *flooding* and should appear only during the initial protocol hand-shakes between two communicating hosts.

A.6.3 Learning process

The records in the MAC table are populated dynamically through a process called *learning*. Consider a frame with *source* address Y entering the switch via port B. If the address Y is not found, then a new entry is added specifying that all frames which have the *destination* equal to Y should be forwarded to the port B. The switch assumes that if it has received frames from Y via port B then the frames *to* Y should also go towards the same port. In the case of an *uplink* between two switches S_1 and S_2 connected via ports A and B respectively, the MAC table for port S_1/A will contain records for each of the addresses known by switch S_2. If switch S_1 receives a frame for a destination reachable from S_2, then it will forward this frame to the uplink, on port A.

A.6.4 A network without loops

In order for the above mechanism to function correctly, the Ethernet network cannot contain loops. In the event of a broadcast frame (a frame that is sent to all the hosts in the network), a loop will generate a *broadcast storm*. The first switch that gets the frame will copy and distribute it to all of its ports. If there is a loop somewhere in the network then these copied frames may return and will be duplicated again, in a chain reaction. To avoid problems like this, there exists the *Spanning Tree Protocol* (STP). Switches use this protocol to communicate periodically and to detect any loops which are created intentionally or by accident. If a loop is found, then one of its link segments is marked as "not active". See [64] and [65] for details about the Spanning Tree Protocol[9].

A.6.5 Virtual LANs

An extension of the Ethernet standard introduces *Virtual Local Area Networks* or VLANs. This permits the partition of the network into multiple sub-networks which can be regarded as independent. Traffic from one VLAN cannot go into another VLAN, unless special settings are done in the switches. With VLANs, loops may appear in an Ethernet network. Inside the same VLAN loops are still illegal. In a VLAN environment, switches maintain MAC tables for each VLAN separately.

[9] A common design technique to assure redundancy for a connection between two switches is to add two or more *backup links* (i.e. creating loops) and to enable the Spanning Tree Protocol. If the primary link fails then STP will revert to a backup link and the connectivity will not be affected.

Appendix B

Components of the GETB card

In this appendix we give additional details about the hardware and firmware components which are used on the GETB card (Chapter 3, Section 3.2, page 50). Block diagrams of the FPGA firmware are presented in Section 3.3.2 and in Figures 3.6 and 3.11.

B.1 Ethernet PHYs

The PHY handles the communication at the Ethernet physical layer. It is the interface between the MAC inside the FPGA and the RJ45 line connectors on the card. The GETB uses a Marvell 88E1111 Alaska Ultra PHY chip [66]. This device supports the standards for 10/100/1000 Gbit/s operation. A block diagram of the PHY is shown in Figure B.1.

The PHY chip communicates with the MAC core inside the FPGA using 2 different channels - one is for data - the MII/GMII interface[1], and one for management - the MDIO interface[2]. The data path has two distinct modes of operation: for 100 Mbit/s Ethernet it uses the MII protocol, while for 1 Gbit/s it uses GMII. MII uses 4-bit words clocked at 25 MHz. GMII uses 8-bit words at 125 MHz. The management path, the MDIO, is designed to be a slow, but reliable, interface – it sends and receives data serially on only 2 wires – the clock MDC (driven from the MAC) and the bidirectional data line MDIO. The connection between the MAC and the PHY is shown in Figure B.2.

The PHY has internal registers for configuration (they can be accessed via MDIO) – the auto-negotiation options can be set, the link speed, and so on. In addition, the PHY has special pins which are supposed to be connected to the LEDS on the RJ45 connector. In the GETB card these LED pins are connected directly to the FPGA. The Handel-C code uses these pins to read the state of the link. From the FPGA, the signals are also routed to the RJ45 LEDs.

[1] MII = Media Independent Interface. GMII = Gigabit MII.
[2] MDIO = Management Data Input/Output. MDC = Management Data Clock.

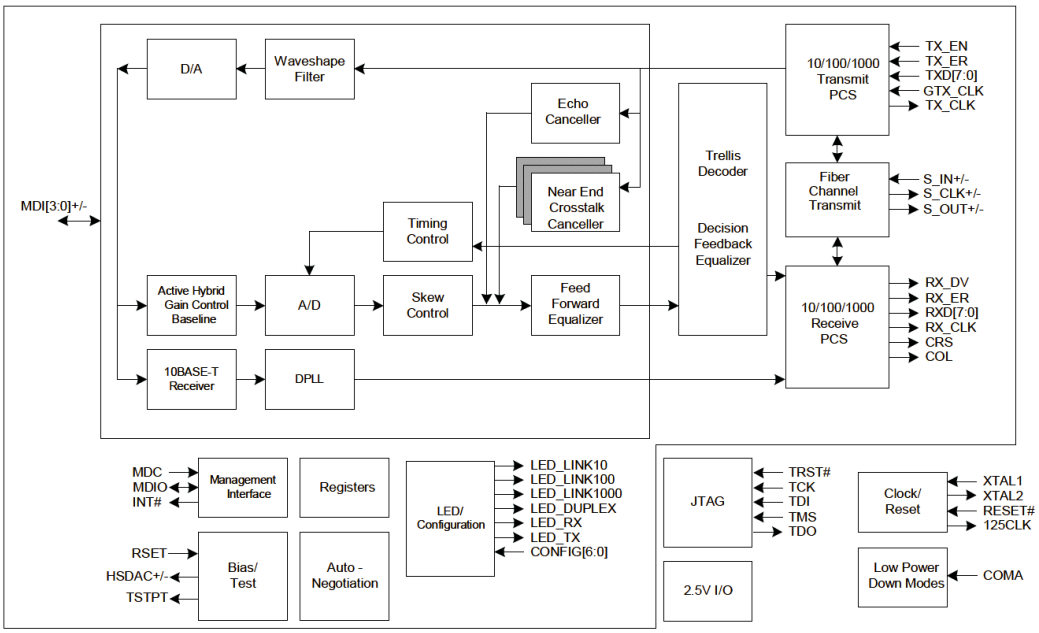


Figure B.1: Ethernet PHY block diagram.

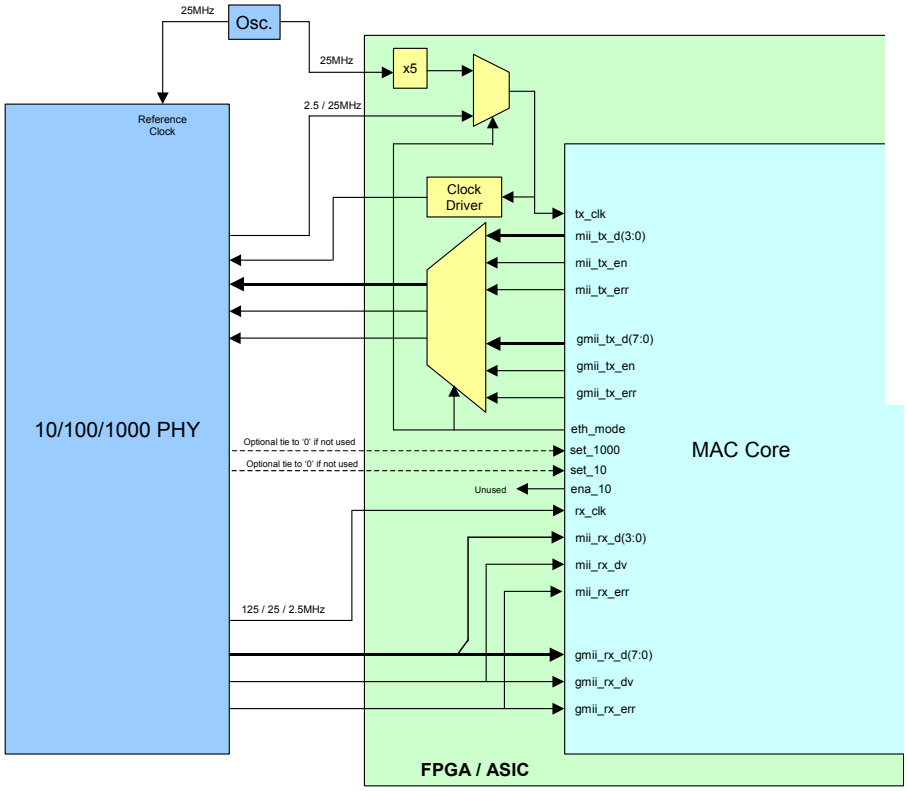


Figure B.2: MAC to PHY connection.

B.2 Ethernet MAC

The MAC is responsible for the creation of Ethernet frames that are sent to the PHY chips. We are using a "software" MAC - the Gigabit Ethernet MAC core from MoreThanIP [32]. The GETB uses two instances of the MAC core (one for each Ethernet port). A block diagram of the MAC is shown in Figure B.3. The core works with multiple clocks:

- A 125 MHz clock for the data path between the MAC and the PHY (the MII/GMII interface).
- A client clock for the Handel-C ETH_main block (41 MHz).
- A clock for the register interface – 25 MHz. From this clock the MAC derives the MDC clock for MDIO transactions with the PHY.

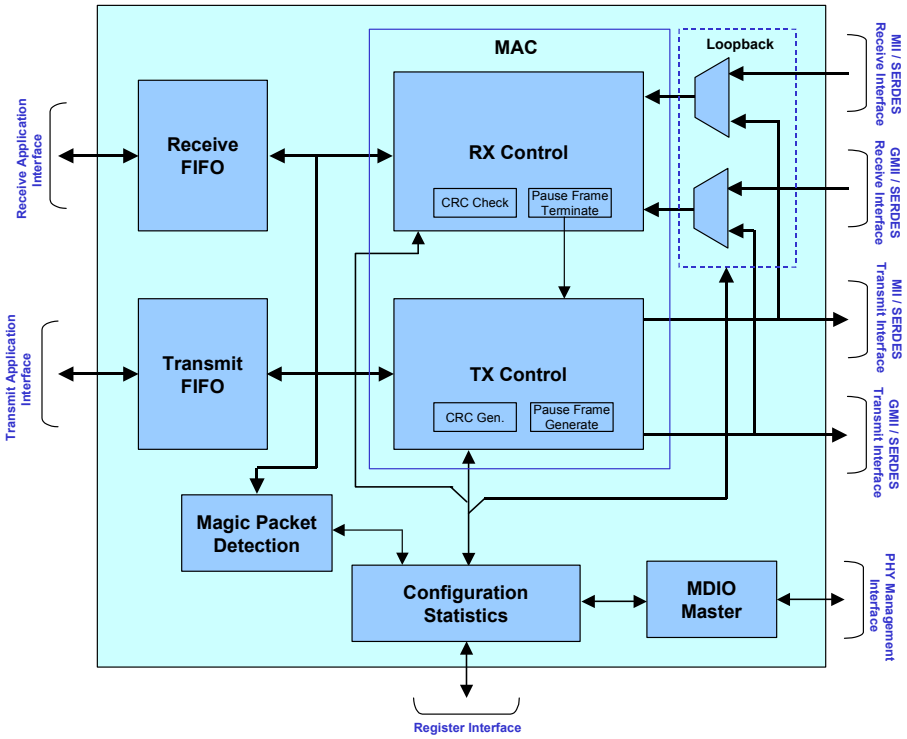


Figure B.3: The MAC IP core.

The communication between the Handel-C application (the client) and the MAC is done using two FIFOs[3] (one for TX and one for RX). Their depth is set when configuring the MAC core and their width is set to 32 bits.

In order to send one packet, the client sets the *ff_tx_sop* signal to 1 (start-of-packet) and starts writing data to the TX FIFO (see Figure B.4). Each time we write something to the FIFO we

[3]FIFO = First In First Out (a queue).

have to check if the MAC is ready to receive the data using the *ff_tx_rdy* signal. Because the MAC transmits one byte per 125 MHz clock cycle and we write 4 bytes per 41 MHz clock cycle, the TX FIFO becomes full quite often. When it is full, the *ff_tx_rdy* will be low and the Handel-C code will pause the transmission and resume when the ready signal is again high (we wait for the TX FIFO to have enough space to receive new data).

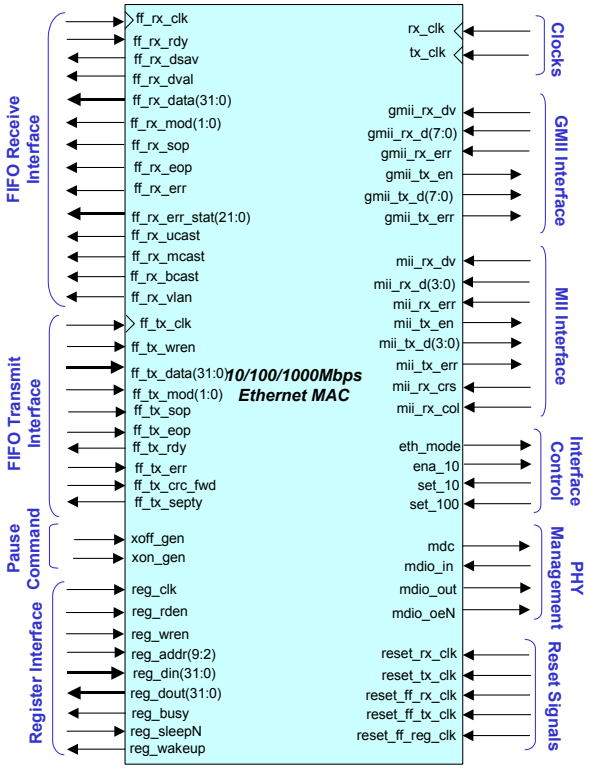


Figure B.4: MAC Interface.

For the receive part we need to check when the RX FIFO contains some valid data that can be read. This is done using the signals *ff_rx_dsav* and *ff_rx_dval*. When the data is available, we are allowed to raise the *ff_rx_rdy* and then we can read the data from the port *ff_rx_data*. If the Handel-C client asserts *ff_rx_rdy* to 1 then the MAC will change the data at the head of the FIFO at each client clock cycle – if the client is busy, it should de-assert the *ff_rx_rdy* signal.

The MAC has an MDIO interface to the PHY (used for control). The MDIO interface works on 2 lines: the bidirectional MDIO line and the clock, MDC. The MAC provides 2 separate data lines: *mdio_in* and *mdio_out* ; plus an output-enable signal *mdio_oeN*. These signals are connected in the top-level VHDL entity of the FPGA to a three-state driver that has one bidirectional output that goes to the MDIO data pins of the chip. If the PHY MDIO is properly connected to the MAC then the PHY registers are mapped onto special MAC registers. Whenever the MAC core detects a read/write to these mapped registers, it will generate an MDIO transaction.

The MAC core expects to receive valid Ethernet frames from the client application. It ensures that the minimum inter-frame gap is respected on the Ethernet line and also computes the CRC checksum for each frame. But is the responsibility of the user application to build valid Ethernet

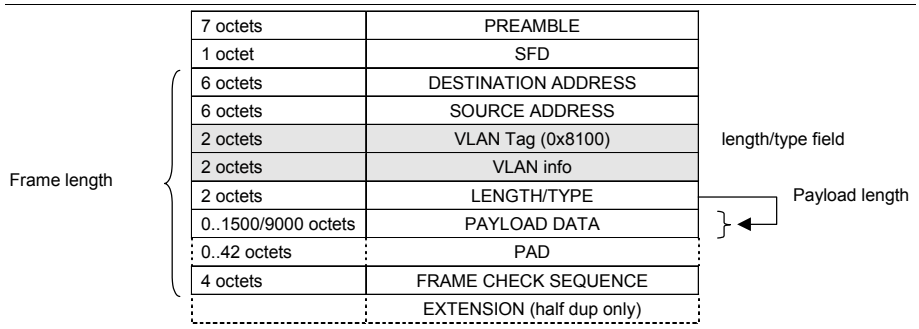


Figure B.5: Ethernet frame format.

frames, that are formatted according to Figure B.5. The MAC keeps counters for the number of frames sent and received, the number of bytes, errors, etc. All these counters are accessible in the MAC registers. The MAC can be configured in promiscuous mode to accept any Ethernet frame and has full support for the Flow-Control congestion handling mechanism.

B.3 GPS clock synchronization

The GETB card was designed to support a global clock synchronization system using a GPS[4] card. We are using a Meinberg GPS card [67]. The card has a 9 pin serial connector that outputs a 1 Hz Pulse per Second (PPS) signal and a 10 MHz signal. The 1 Hz signal is synchronized with the GPS satellites – this is the basis of the synchronization system.

The two signals, 1 Hz and 10 MHz, need to be distributed to all GETB cards on the GPS RJ45 connector. In stand-alone mode this is done using a special serial-to-RJ45 cable. When multiple cards are used, a GPS fan-out box distributes the signals to all cards. The two GPS signals are in TTL standard and they need to be converted to LVDS. This is done either using a circuit embedded in the connector, or in the GPS fan-out box. The FPGA does not accept TTL inputs (the FPGA I/O pins work at 3.3V, TTL is 5V). The GPS system has been tested only in stand-alone mode due to the unavailability of the fan-out box.

The GPS RJ45 connector is used to receive the clocks from the GPS card or from the GPS fan-out box. There are 4 signals used on this connector:

1 Hz PPS (INPUT) – A pulse of duration 0.2 seconds that is sent by the GPS card at the beginning of each second – this signal is synchronized with the UTC time[5].

10 MHz clock (INPUT) – This is coming from an oscillator on the GPS card.

AUX 0 (OUTPUT) – Used for clock synchronization. Connected only to a card which was assigned the role of "GETB master". A pulse is sent by the GETB master card to all the other cards to trigger various actions.

[4]GPS = Global Positioning System.
[5]UTC = Coordinated Universal Time.

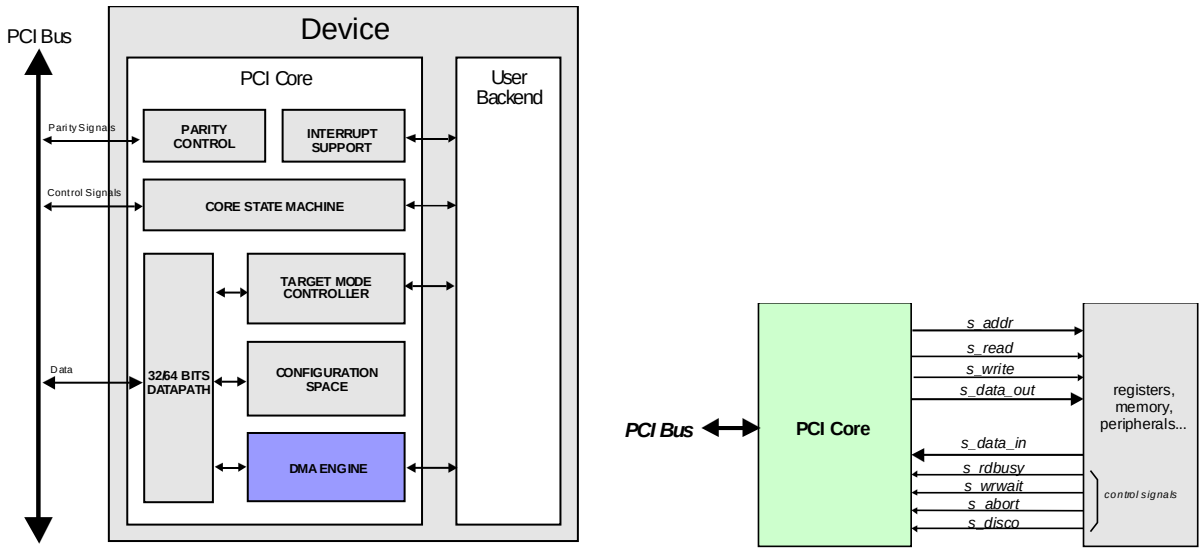
AUX 1 (INPUT) – Connected to the AUX 0 output from the master card – each GETB card should receive a copy of the AUX 0 from the master, on the AUX 1 pin.

The 10 MHz and 1 Hz signal are used in LVDS differential mode. The AUX 1 and AUX 0 cannot be used in differential mode because of the placement of the pins on the FPGA (they are too close to pins which use single-ended transmission). Because of this, we observed crosstalk between the AUX signals and the GPS clocks (the clocks are corrupted when something happens on the AUX lines).

B.4 PCI Interface

The GETB card complies with the PCI 2.2 standard. The PCI connector is made for 64 bit transfers, but we are using only the 32 bit interface. The PCI communication is handled by a PCI IP Core coming from PLD Applications [29]. This PCI core supports 66 MHz, but we can only run it at 33 MHz because of speed limitations in our version of the Stratix FPGA.

The PCI protocol is handled by the PCI core. In Figure B.6 we show a block diagram of the PCI core and the signals that are seen by the back-end. In our case the back-end is the PCI_main Handel-C block (see Section 3.2.3.2).



(a) Architecture

(b) Interface to the back-end logic

Figure B.6: The PCI core.

The PCI_main Handel-C block talks to the PCI core and translates PCI requests into Handel-C function calls. This allows us to access all the memory available on the card from the host computer. Any PCI device defines, when it is initialized, a set of memory regions that are visible to the host. The GETB card advertises the following regions:

Region 0 (length = 128 bytes) – A special region used to send commands to the card and read status registers and counters from the Handel-C code.

Regions 1 and 2 (length = 512 Kbyte) – These regions are mapped to the two SRAM memories available on the GETB card.

Regions 3 and 4 (length = 64 Mbyte) – They are mapped to the SDRAM memories on the card.

The figure below shows how the GETB card is seen by a Linux computer:

```
$ /sbin/lspci -v -d 10dc:0313
03:01.0 Ethernet controller: CERN/ECP/EDU: Unknown device 0313
    Flags: bus master, slow devsel, latency 64, IRQ 24
    Memory at f0300000 (32-bit, non-prefetchable) [size=128]    # registers
    Memory at f0280000 (32-bit, non-prefetchable) [size=512K]   # SRAM 0
    Memory at f0200000 (32-bit, non-prefetchable) [size=512K]   # SRAM 1
    Memory at f8000000 (32-bit, non-prefetchable) [size=64M]    # SDRAM 0
    Memory at f4000000 (32-bit, non-prefetchable) [size=64M]    # SDRAM 1
    Capabilities: <available only to root>
```

B.5 The Handel-C language

Handel-C [34] is a language which aims to allow software engineers to design hardware. The language is based on ANSI-C, but was extended with special hardware-specific constructs. It comprises all common expressions necessary to describe complex algorithms, but lacks processor-oriented features like pointers and floating point arithmetic. Figure B.7 shows a feature comparison between ANSI C and Handel-C.

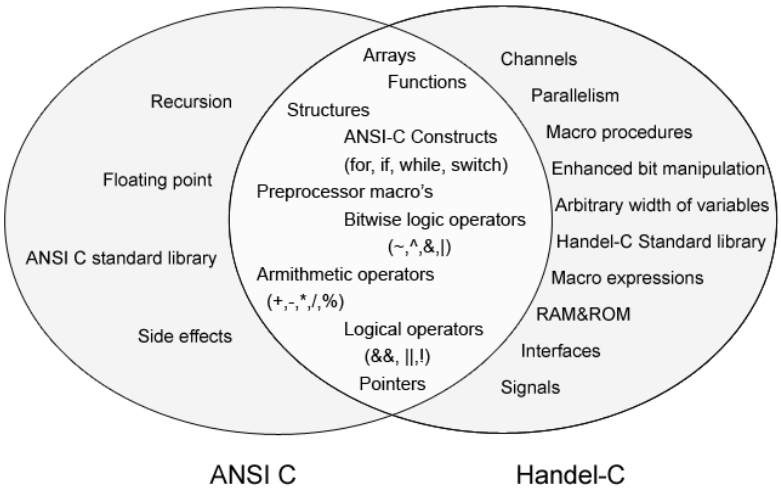
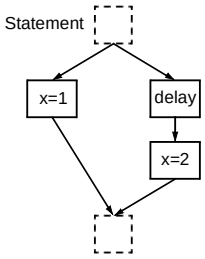


Figure B.7: Handel-C vs. ANSI C (Feature comparison).

Being targeted to hardware, the language provides a few optimizing features. For example the user can specify the bit-width of integer variables and can access I/O ports. In addition, Handel-C has built-in support for parallel constructs and synchronization primitives (channels, semaphores). By using the *par* statement the user can say which code blocks run in parallel. The timing model in Handel-C is very simple – the basic rule is that any assignment takes one clock cycle and everything else is combinatorial logic which does not consume clock cycles. Handel-C is a *cycle-accurate* language – the programmer can tell in advance how many cycles a code block will require.

```
par
{
  x = 1;
  {
      delay;
      x = 2;
  }
}
```



(a) Code (b) Diagram

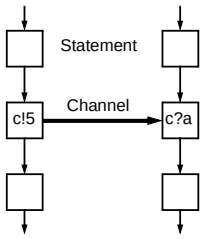
Figure B.8: Handel-C – Parallel statements.

Any Handel-C program can be synthesized into a netlist. The company Celoxica sells a complete design environment which includes an editor and a simulator. The simulation environment is similar to a standard debugger and is more easier to use than standard HDL[6] simulation packages (it supports breakpoints, watches, step-by-step execution, etc). Here we present just some very simple pieces of code for parallel statements and channels.

In Figure B.8 we see an example of parallel code block and its associated block diagram. The execution of Handel-C statements is sequential, unless a *par* block is used. All statements in a *par* block are implemented on dedicated hardware resources (there are no time sharing and scheduling mechanisms like for processes / threads running on a standard CPU). The execution will continue with the statements after the *par* block only when all the parallel tasks are finished. The *par* block is equivalent to the *fork* and *join* system calls in Unix-like operating systems.

[6]HDL = Hardware Description Language.

```
unsigned int 8 a;
chan unsigned int 8 c;
par {
       c ! 5;
       c ? a;
}
```



(a) Code (b) Diagram

Figure B.9: Handel-C – Channels.

Figure B.9 presents the use of channels. Channels are used for communicating between parallel branches of code. One branch writes to a channel and a second branch reads from it. The communication only occurs when both tasks are ready for the transfer, at which point one item of data is transferred between the two parallel branches. If any of the two end-points of the channel is not ready then the other process will wait. The syntax for channel write is *channel ! output_data*, while for channel read is *channel ? input_register*.

Appendix C

Mathematical background on statistical sampling

In this appendix we explain the theory behind packet sampling. It complements the information from Chapter 6 about the *sFlow* standard. We begin by giving a statement of the problem we want to solve[1].

Suppose we have a system that can transfer and sample packets. We let it run for a a given time period during which we count a total of N packets. Among these packets there are N_c which have an interesting property or attribute (which we call *attr*). The number N_c of interesting packets is unknown. We sample at random a number of n packets out of the total of N. From these samples we count c packets which have the interesting property *attr*. The aim is to find an estimate of the number N_c, which we shall call $\hat{N}_c$. We would also like to evaluate the *accuracy* of this estimation. The main notations that will be used are summarized in the table below:

Symbol	Meaning	Comments
N	Total number of packets counted during the period of observation	Measured
n	Number of sampled packets taken from the total of N	Measured
c	Number of sampled packets having the attribute of interest *attr*	Measured
N_c	Total number of packets which have the attribute of interest *attr*	Unknown

The problem can be solved by common sense: if there are c out of n interesting packets among the samples then it is very likely that the same ratio will be kept for the whole set of packets, so there should be approximately:

$$\hat{N}_c = \frac{c}{n} \cdot N$$

interesting packets in the total of N. In the next pages we'll show that this is indeed the case

[1]The contents of this section is partially based on [68] and [56].

and we shall provide a quantitative measure for the error that is expected from the measurement (the so called *confidence interval*).

In order to make inferences about N_c we shall consider the number of interesting samples, c, a random variable. This variable describes the number c of *good* events (interesting packets) in a sequence of n draws (the number of samples). These draws are taken from a total population of N events, out of which there are exactly N_c which are *good*. This description corresponds to the definition of the *Hyper-geometric distribution* [69]. For a random variable with this distribution, the probability of having a certain value c is:

$$P_{hyper}(c; N, N_c, n) = \frac{\binom{N_c}{c}\binom{N-N_c}{n-c}}{\binom{N}{n}}$$

where $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ is the number of possible ways to choose k "successes" from n observations.

The formula gives the probability that in n packet samples we shall find c which have our interesting attribute. If the total number of packets N (the population) is much larger than the number of samples n, then the *Hyper-geometric distribution* can be approximated by a *Binomial distribution* with the parameters n (number of trials) and $p = \frac{N_c}{N}$ (the probability of "success" in a single trial)[2]. In this case the probability that we shall find c interesting packets will be:

$$P_{binomial}(c; n, p) = \binom{n}{c} p^c (1-p)^{n-c}$$
$$\text{with} \quad p = \frac{N_c}{N}$$

The binomial distribution [70], [71] describes the behavior of a count variable, c, if:

- The number of observations n is fixed.
- Each observation is independent.
- Each observation represents one of two outcomes ("success" or "failure").
- The probability of "success", p, is the same for each observation.

We have the following mean μ_c and variance σ_c^2 for the binomial distribution which characterizes the number of interesting packets, c:

[2]The Hyper-geometric distribution $P_{hyper}(c)$ of parameters (N, N_c, n) can be seen as the sum of n Bernoulli random variables (a Bernoulli variable has value 1 with probability p and value 0 with probability $1-p$). The i^{th} variable takes value 1 if sample number i is a "good" sample and 0 otherwise (the sum of all these variables gives c). But unlike in the Binomial distribution, the n Bernoulli variables in the Hyper-geometric distribution *are not independent*. The probability of "success" at sample i depends on how many successes were among samples $1..(i-1)$. If the number of samples n is small relative to the total population N, then the probabilities will vary slightly for different values of i. So in this case the n variables can be seen as almost independent, all of them having the same probability of success, $p = \frac{N_c}{N}$. Then the Hyper-geometric distribution can be approximated by a Binomial distribution for n independent observations of a Bernoulli random variable with success probability p.

$$E(c) = \mu_c = np$$
$$\sigma_c^2 = np(1-p)$$

The parameter $p = \frac{N_c}{N}$ of the distribution of c is the *true proportion* (or *population proportion*) of the interesting packets among the whole set and is the unknown in our problem. If we had p then we could calculate $N_c = p \cdot N$. In the following we shall find an *estimate* for p which we shall call P. Once having the estimate of p, we can also estimate the total number of interesting packets as $\hat{N}_c = P \cdot N$.

We shall use the *sampling proportion*, defined as:

$$P = \frac{c}{n}$$

as an estimate for p (the value of P can be computed using the sample data). The proportion P is another random variable which depends on the random variable c. Then the mean value of P will be:

$$E(P) = \mu_P = E\left(\frac{c}{n}\right) = \frac{\mu_c}{n} = \frac{np}{n} = p$$

The fact that $E(P) = p$ means that $P = \frac{c}{n}$, the sample proportion, is an *unbiased estimator* for the population proportion p ([72] page 265). The variance of the random variable P will be:

$$\sigma_P^2 = \sigma^2\left(\frac{c}{n}\right) = \frac{\sigma_c^2}{n^2} = \frac{np(1-p)}{n^2} = \frac{p(1-p)}{n}$$

So if the number of samples, n, increases then the variance of the sample proportion, P, decreases and the parameter p of the binomial distribution of c will be better estimated.

The variable P can be regarded as the average value of a sum of n Bernoulli random variables (these variables have mean p and variance $p(1-p)$). Each such variable corresponds to a packet sample and takes value 1 if the sample is good and 0 if it isn't. Then by the *Central Limit Theorem* [73] we have that for large values of n, the distribution of P will tend to a *normal distribution* having the same mean as any of the "member" Bernoulli variables and the variance n times smaller. So we have that:

$$\text{For large n:} \quad P \to P_{normal}\left(c; \mu = p, \sigma^2 = \tfrac{p(1-p)}{n}\right)$$

Using these approximations, the *maximum likelihood estimate*[3] of p (the population proportion and the mean value for the normal distribution characterizing P) is the sample proportion, $P = \frac{c}{n}$.

[3]Let $f(x;\theta)$ be a probability density for which we wish to estimate the value of the unknown parameter θ using a set of sample values $x_1, \ldots, x_n$. We define the *likelihood function* $L(x_1, ..x_n; \theta) = f(x_1;\theta) \cdot \ldots \cdot f(x_n;\theta)$. The maximum likelihood estimate (MLE) of θ is $\hat{\theta}$ which maximizes the function L. For a normal distribution with unknown mean, the MLE is the average value of the samples. For more details see [72] page 287 and [74].

An *unbiased* estimate[4] for the variance of P is the following:

$$S^2 = \frac{P(1-P)}{n-1} = \frac{\frac{c}{n}(1-\frac{c}{n})}{n-1} = \frac{c(1-\frac{c}{n})}{n(n-1)}$$

So we finally have an estimate for the population proportion and we also have information about the variance of this estimate. As we made the assumption that the sample proportion P is normally distributed, we can calculate the *confidence interval* for the estimate of p.

A confidence interval gives an estimated range of values which is likely to include an unknown population parameter (which is p in this case), the estimated range being calculated from a given set of sample data[5].

For a normally distributed random variable of mean μ and variance σ^2 the probability that it takes values in a certain interval centered around the mean is given by [75]:

$$P(\mu - z_\alpha\sigma < x < \mu + z_\alpha\sigma) = erf\left(\frac{z_\alpha}{\sqrt{2}}\right)$$
$$\text{where} \quad erf(z) = \frac{2}{\sqrt{\pi}}\int_0^z e^{-t^2}dt$$

For example, if $z_\alpha = 1.96$ then the above probability is 0.95. So 95% of the values will be at distance $1.96 \cdot \sigma$ from the mean μ.

In our problem we don't know the exact values of μ and σ^2 so we shall use their estimates: P and S^2. We can express the confidence interval for the estimate $P = \frac{c}{n}$ of the population proportion. For example we are 95% confident that the observed values of P are in the interval below which contains also the true (unknown) value of the parameter p:

$$\left[\frac{c}{n} - 1.96 \cdot S,\ \frac{c}{n} + 1.96 \cdot S\right]$$

The width of this interval depends on the variance estimate, S, which is inversely proportional to the number of samples, n (the interval can be made narrower if we take more samples).

However, we are more interested in finding an estimate for the total number of good packets, N_c. This estimate is:

$$\hat{N}_c = P \cdot N = \frac{c}{n} \cdot N$$

[4]For a normal distribution $N(\mu, \sigma^2)$ the maximum likelihood estimate for the variance using a set of samples x_i is $\hat{\sigma^2} = \frac{\sum (x_i - \hat{\mu})^2}{n}$, where $\hat{\mu}$ is the MLE for the mean [74]. This estimate is *biased* because $E(\hat{\sigma^2}) \neq \sigma^2$. An *unbiased* estimate is S^2 as defined in the text (see also [72] page 262).

[5]The width of the confidence interval gives us some idea about how uncertain we are about the unknown parameter (the precision). A very wide interval may indicate that more data should be collected before anything very definite can be said about the parameter.

The estimated variance of this parameter is $N^2 \cdot S^2$. Therefore the 95% confidence interval for $\hat{N}_c$ is:

$$\left[\frac{c}{n} \cdot N \pm 1.96 \cdot N \cdot S\right]$$
$$\text{with} \quad S^2 = \frac{c(1-\frac{c}{n})}{n(n-1)}$$

So the maximum relative estimation error will be:

$$error = \frac{1.96 \cdot N \cdot S}{\hat{N}_c} = \frac{1.96 \cdot N \cdot \sqrt{\frac{c(1-\frac{c}{n})}{n(n-1)}}}{\frac{c}{n} \cdot N}$$

This reduces to:

$$error = 1.96 \cdot \sqrt{\left(\frac{1}{c} - \frac{1}{n}\right) \cdot \left(\frac{n}{n-1}\right)}$$

When the number of samples n is large, the error simplifies to (now written in percents):

$$error\% \approx 196 \cdot \sqrt{\frac{1}{c}}$$

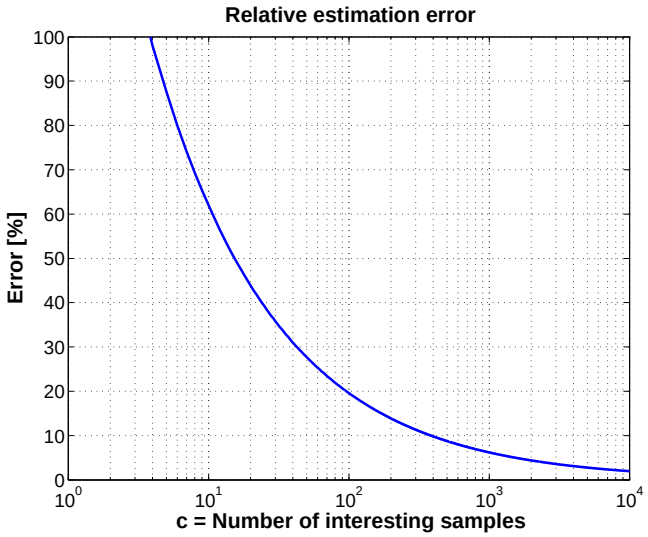


Figure C.1: Statistical sampling – Relative sampling error.

It can be observed that the accuracy of the measurements doesn't depend on the total number of packets, but on the number of samples used to make the measurement. The Figure C.1 shows the plot of the relative estimation error as a function of the number of interesting packets.

The estimate $\hat{N}_c$ gives the number of packets in the class c. The number of bytes consumed by this particular class can be computed as follows:

$$\hat{B}_c = \bar{b_c} \cdot \hat{N}_c$$

where $\bar{b_c}$ is the average size of a packet sample taken from class c:

$$\bar{b_c} = \frac{\sum_{i=1}^{c} b_{ci}}{c}$$

In order to increase the accuracy of the estimation, there are two possibilities: either we increase the sampling rate, either we take samples for a longer period of time. In practice, statistical sampling is used to find the sources of traffic consuming the most of the network bandwidth so that the number c is usually large enough to make quite precise measurements.

List of Figures

Bibliography

[1] "CERN – The European Organization for Nuclear Research." [Online]. Available: http://www.cern.ch/

[2] ATLAS Collaboration, "ATLAS Detector and Physics Performance Technical Design Report," *CERN/LHCC 99-14*, May 1999. [Online]. Available: http://atlas.web.cern.ch/Atlas/GROUPS/PHYSICS/TDR/access.html

[3] S. Stancu, "Networks for the ATLAS LHC Detector: Requirements, Design and Validation," Ph.D. dissertation, Universitatea "Politehnica" Bucureşti, 2005.

[4] S. Stancu, M. Ciobotaru, and D. Francis, "Relevant features for DataFlow switches," April 2005. [Online]. Available: http://cern.ch/ciobota/papers/2005_switch_features.pdf

[5] ATLAS HLT/DAQ/DCS Group, "ATLAS High-Level Trigger Data Acquisition and Controls Technical Design Report," *CERN/LHCC/2003-022*, October 2003. [Online]. Available: http://atlas-proj-hltdaqdcs-tdr.web.cern.ch/atlas-proj-hltdaqdcs-tdr/

[6] J. Bystricky, D. Calvet, M. Huet, P. L. Du, and I. Mandjavidze, "Studies of ATM for ATLAS High Level Triggers," in *3rd Workshop on Network-based Event Building - DAQ 2000*, Lyon, France, October 2000.

[7] M. Dobson, "The Use of Commodity Products in the ATLAS Level-2 Trigger," in *Computing in High Energy and Nuclear Physics*, Padova, Italy, 2000. [Online]. Available: http://chep2000.pd.infn.it/short_p/spa_b035.pdf

[8] The CMS Collaboration, "The CMS Data Acquisition and High-Level Trigger Technical Design Report (Volume 2)," *CERN/LHCC 02-26 CMS TDR 6*, 2002. [Online]. Available: http://cmsdoc.cern.ch/cms/TRIDAS/Temp/CMS_DAQ_TDR.pdf

[9] G. Antchev, E. Cano, S. Cittolin, S. Erhan, B. Faure, D. Gigi, J. Gutleber, C. Jacobs, F. Meijers, E. Meschi, A. Ninane, L. Orsini, L. Pollet, A. Racz, D. Samyn, and N. S. and, "Evaluation of Myrinet for the Event Builder of the CMS experiment," 2000. [Online]. Available: http://citeseer.ist.psu.edu/antchev00evaluation.html

[10] F. Saka, "Ethernet for the ATLAS Second Level Trigger," Ph.D. dissertation, Royal Holloway College, Physics Department University of London, December 2000.

[11] R. Dobinson, S. Haas, K. Korcyl, M. J. LeVine, J. Lokier, B. Martin, C. Meirosu, F. Saka, and K. Vella, "Testing and Modeling Ethernet Switches and Networks for Use in ATLAS High-level Triggers," in *IEEE Trans. on Nuclear Science*. IEEE Nuclear and Plasma Sciences Society, 2001, vol. 48, No. 3, pp. 607–612.

[12] R. Dobinson, M. Dobson, S. Haas, B. Martin, M. J. LeVine, and F. Saka, "IEEE 802.3 Ethernet, Current Status and Future Prospects at the LHC," *CERN open-2000-311*, October 2000. [Online]. Available: http://doc.cern.ch/archive/electronic/cern/preprints/open/open-2000-311.pdf

[13] R. Dobinson, K. Korcyl, and M. LeVine, "An ATLAS TDAQ Candidate architecture," *DC note 049*, July 2002. [Online]. Available: https://edms.cern.ch/document/391584

[14] F. Barnes, R. Beuran, R. Dobinson, M. J. LeVine, B. Martin, J. Lokier, and C. Meirosu, "Testing Ethernet networks for the ATLAS data collection system," in *IEEE Trans. on Nuclear Science*. IEEE Nuclear and Plasma Sciences Society, April 2002, vol. 49, No. 2, p. 516.

[15] H. Beck, R. Dobinson, K. Korcyl, and M. LeVine, "ATLAS TDAQ: A Network-based Architecture," *DC note 059*, February 2003. [Online]. Available: https://edms.cern.ch/file/391592/2.2/DC-059.pdf

[16] S. Stancu, M. Ciobotaru, B. Dobinson, K. Korcyl, and E. Knezo, "The use of Ethernet in the Dataflow of the ATLAS Trigger and DAQ," in *Computing in High Energy and Nuclear Physics*, La Jolla, California, 2003. [Online]. Available: http://cern.ch/ciobota/papers/2003_ethernet_for_atlas.pdf

[17] M. Ciobotaru, S. Stancu, M. J. LeVine, and B. Martin, "GETB, a Gigabit Ethernet Application Platform: its Use in the ATLAS TDAQ Network," in *IEEE Trans. on Nuclear Science*. IEEE Nuclear and Plasma Sciences Society, 2006, vol. 53, No. 3, pp. 817–825. [Online]. Available: http://cern.ch/ciobota/papers/2005_getb_stockholm.pdf

[18] S. Stancu, M. Ciobotaru, and K. Korcyl, "ATLAS TDAQ DataFlow Network Architecture Analysis and Upgrade Proposal," in *IEEE Trans. on Nuclear Science*. IEEE Nuclear and Plasma Sciences Society, 2006, vol. 53, No. 3, pp. 826–833. [Online]. Available: http://cern.ch/ciobota/papers/2005_atlas_net_proposal.pdf

[19] S. Stancu, M. Ciobotaru, C. Meirosu, L. Leahu, and B. Martin, "Networks for the ATLAS Trigger and Data Acquisition," in *Computing in High Energy and Nuclear Physics*, Mumbai, India, 2006. [Online]. Available: http://cern.ch/ciobota/papers/2006_networks_tdaq.pdf

[20] T. Sridhar, "Redundancy: Choosing the Right Option for Net Designs," July 2004. [Online]. Available: http://www.commsdesign.com/showArticle.jhtml?articleID=25600515

[21] L. M. S. Committee, "IEEE Std 802.3ad – Aggregation of Multiple Link Segments," IEEE Computer Society.

[22] "The Python Programming Language." [Online]. Available: http://www.python.org/

[23] "perl – Practical Extraction and Report Language." [Online]. Available: http://www.perl.org/

[24] Ixia, "Ixia Performance Testing." [Online]. Available: http://www.ixiacom.com/products/performance_applications/

[25] Spirent, "Smartbits." [Online]. Available: http://www.spirentcom.com/

[26] B. Martin, M. LeVine, and M. Ciobotaru, "A new Gigabit Ethernet tester," CERN, Tech. Rep. ATL-DQ-ES-0012, May 2003. [Online]. Available: https://edms.cern.ch/file/386279/2/newGigTester.pdf

[27] Wikipedia, "Network processor." [Online]. Available: http://en.wikipedia.org/w/index.php?title=Network_processor&oldid=124536356

[28] M. Ciobotaru, "An introduction to Field Programmable Gate Arrays (PhD Report)," 2005. [Online]. Available: http://cern.ch/ciobota/papers/2005_phdrep_fpga_intro.pdf

[29] "PLD Applications PCI Core." [Online]. Available: http://www.plda.com/prodetail.php?pid=38

[30] Intel Corporation, "Intel IXF1002 dual-port Gigabit Ethernet MAC." [Online]. Available: http://www.intel.com/design/network/products/lan/controllers/IXF1002.htm

[31] LSI Corporation, "LSI 8101/8104 Gigabit Ethernet Controller." [Online]. Available: http://www.lsi.com/files/docs/techdocs/networking/8101_8104_DS.pdf

[32] "MoreThanIP Gigabit Ethernet MAC Core." [Online]. Available: http://www.morethanip.com/mbps_10_100_1000_mac.htm

[33] Altera Corporation, "SDR SDRAM Controller Reference Design." [Online]. Available: http://www.altera.com/support/refdesigns/sys-sol/computing/ref-sdr-sram.html

[34] Celoxica, "The Handel-C Programming Language." [Online]. Available: http://www.celoxica.com/technology/c_design/handel-c.asp

[35] M. Ciobotaru, "VHDL_Gen – Creating hardware using Python." [Online]. Available: http://cern.ch/ciobota/project/vhdl_gen/

[36] "AN 122 – Using Jam STAPL for ISP and ICR via an Embedded Processor," 2003. [Online]. Available: http://www.altera.com/literature/an/an122.pdf

[37] Wikipedia, "Occam (programming language)," 2007. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Occam_%28programming_language%29&oldid=150957478

[38] M. Joos, "IO_RCC – A package for user level access to I/O resources on PCs and compatible computers," CERN, Tech. Rep. ATL-D-ES-0008, October 2003. [Online]. Available: https://edms.cern.ch/document/349680/2

[39] Shilad Sen (Sourcelight Technologies Inc.), "A Fast XML-RPC Implementation." [Online]. Available: http://sourceforge.net/projects/py-xmlrpc/

[40] M. Ivanovici, "Network quality degradation emulation – An FPGA-based approach to application performance assessment," Ph.D. dissertation, Universitatea "Politehnica" Bucureşti, January 2006.

[41] "X.700 – Management framework for open systems interconnection for CCITT applications," 1992.

[42] Wikipedia, "Simple Network Management Protocol." [Online]. Available: http://en.wikipedia.org/w/index.php?title=Simple_Network_Management_Protocol&oldid=131715917

[43] "Aprisma Spectrum Network Management System." [Online]. Available: http://www.aprisma.com/

[44] C. Meirosu, B. Martin, A. Topurov, and A. Al-Shabibi, "Planning for Predictable Network Performance in the ATLAS TDAQ," in *Computing in High Energy and Nuclear Physics*, Mumbai, India, 2006.

[45] S. M. Batraneanu, A. Al-Shabibi, M. Ciobotaru, M. Ivanovici, L. Leahu, B. Martin, and S. Stancu, "Operational Model of the ATLAS TDAQ Network," in *Proc. IEEE Real Time 2007 Conference*, Chicago, USA, May 2007. [Online]. Available: http://cern.ch/ciobota/papers/2007_operational_model.pdf

[46] Wikipedia, "Netconf – The Network Configuration Protocol." [Online]. Available: http://en.wikipedia.org/w/index.php?title=Netconf&oldid=123491664

[47] M. Ciobotaru, "sw_script – Unified interface for switch configuration and monitoring." [Online]. Available: http://cern.ch/ciobota/project/sw_script/

[48] Y. Breitbart, M. Garofalakis, B. Jai, C. Martin, R. Rastogi, and A. Silberschatz, "Topology discovery in heterogeneous IP networks: the NetInventory system," *IEEE/ACM Trans. Netw.*, vol. 12, no. 3, pp. 401–414, 2004.

[49] B. Lowekamp, D. O'Hallaron, and T. Gross, "Topology discovery for large Ethernet networks," in *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM Press, 2001, pp. 237–248.

[50] Y. Bejerano, "A Simple And Efficient Topology Discovery Scheme For Large Ethernet LANs," in *Proc. of IEEE INFOCOM*, 2006. [Online]. Available: http://www1.bell-labs.com/user/bejerano/Papers/sklt_tree_L2_net_disc_IC06V2.pdf

[51] AT&T Research, "Graphviz - Graph Visualization Software." [Online]. Available: http://www.graphviz.org/

[52] E. Adar, "GUESS: A Language and Interface for Graph Exploration." [Online]. Available: http://graphexploration.cond.org/chi2006/guess-chi2006.pdf

[53] N. Dawes, D. Schenkel, and M. Slavitch, "Method of determining the topology of a network of objects," U.S. Patent 6,411,997, June 2002.

[54] D. Schenkel, M. Slavitch, and N. Dawes, "Method of determining topology of a network of objects which compares the similarity of the traffic sequences/volumes of a pair of devices," U.S. Patent 5,926,462, July 1999.

[55] R. Black, A. Donnelly, and C. Fournet, "Ethernet Topology Discovery without Network Assistance," in *Proceedings of the 12th IEEE Conference on Network Protocols*, 2004. [Online]. Available: http://www.ieee-icnp.org/2004/papers/9-1.pdf

[56] J. Jedwab, P. Phaal, and B. Pinna, "Traffic Estimation for the Largest Sources on a Network, Using Packet Sampling with Limited Storage," HP Labs, Tech. Rep., 1992. [Online]. Available: http://www.hpl.hp.com/techreports/92/HPL-92-35.html

[57] "RFC 3176 - InMon Corporation sFlow: A Method for Monitoring Traffic in Switched and Routed Networks." [Online]. Available: http://www.faqs.org/rfcs/rfc3176.html

[58] "Cisco IOS NetFlow." [Online]. Available: http://www.cisco.com/en/US/products/ps6601/products_ios_protocol_group_home.html

[59] "sFlow Homepage." [Online]. Available: http://www.sflow.org/

[60] "XML-RPC Homepage." [Online]. Available: http://www.xmlrpc.com/

[61] "sFlow Collectors." [Online]. Available: http://www.sflow.org/products/collectors.php

[62] A. Tanenbaum, *Computer Networks*. Prentice Hall Professional Technical Reference, 2002.

[63] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach Featuring the Internet*. Pearson Benjamin Cummings, 2004.

[64] LAN MAN Standards Committee, "IEEE Std 802.1d Media Access Control (MAC) Bridges," IEEE Computer Society, 2004.

[65] Cisco, "Understanding the Spanning-Tree Protocol." [Online]. Available: http://www.cisco.com/univercd/cc/td/doc/product/rtrmgmt/sw_ntman/cwsimain/cwsi2/cwsiug2/vlan2/stpapp.htm

[66] Marvell, "Alaska Ultra 88E1111 Data sheet." [Online]. Available: http://www.marvell.com

[67] Meinberg Funkuhren, "The GPS167PCI GPS Clock User's manual." [Online]. Available: http://www.meinberg.de

[68] P. Phaal and S. Panchen, "Packet Sampling Basics." [Online]. Available: http://www.sflow.org/packetSamplingBasics/

[69] Wikipedia, "Hypergeometric distribution." [Online]. Available: http://en.wikipedia.org/w/index.php?title=Hypergeometric_distribution&oldid=126062252

[70] Department of Statistics at Yale University, "The Binomial Distribution." [Online]. Available: http://www.stat.yale.edu/Courses/1997-98/101/binom.htm

[71] Wikipedia, "Binomial distribution." [Online]. Available: http://en.wikipedia.org/wiki/Binomial_distribution

[72] T. Soong, *Fundamentals of Probability and Statistics for Engineers*. John Wiley & Sons Ltd., 2004.

[73] Wikipedia, "Central limit theorem." [Online]. Available: http://en.wikipedia.org/w/index.php?title=Central_limit_theorem&oldid=128202893

[74] E. W. Weisstein, "Maximum likelihood," Wolfram MathWorld. [Online]. Available: http://mathworld.wolfram.com/MaximumLikelihood.html

[75] ——, "Confidence Interval," Wolfram MathWorld. [Online]. Available: http://mathworld.wolfram.com/ConfidenceInterval.html